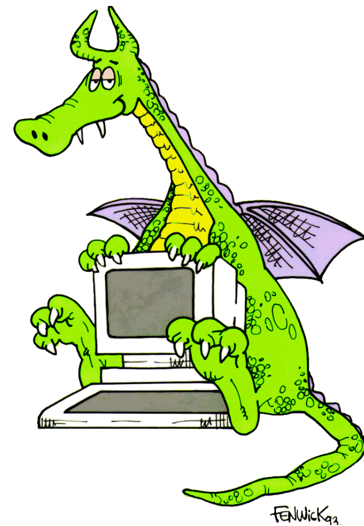
FARGOS Development, LLC

http://www.fargos.net

The attached paper, "*Concurrent Network Management with a Distributed Management Tool*", was written some time ago. Some of the information is as relevant and novel today as it was when published; however, one particular set of information disclosed by the paper has been rendered inaccurate due to the passage of time: none of the authors retained the email and postal addresses listed in the *Author Information* section provided after the paper's conclusions.

Unfortunately, Rob Lehman passed away in June of 2005.

To compensate for this obsolete information, the original content has been overlaid with a strikethrough and current email addresses appear near the margins. No other alterations to the content have been made.



More than merely state of the art...

Concurrent Network Management with a Distributed Management Tool

R. Lehman, G. Carpenter, & N. Hien – IBM T. J. Watson Research Center

ABSTRACT

As distributed computing has become more prevalent, the need to effectively monitor and manage computer networks has grown in importance. In the the TCP/IP world, this has meant monitoring routers, gateways, hubs and other devices using SNMP and other protocols. However, network management stations have tended to be single-use, turn-key applications that lack scalability. In addition, monitoring and measurement have extended beyond the level of routers and gateways to end-user workstations where system administrators are using SNMP to keep track of traditional system management functions using the existing network management framework.

Most existing monitoring and management tools do not scale when the monitored network may include hundreds of routers and thousands of workstations. For large, complex networks, it is impractical to have a central monitoring and data collection point that generates all management queries, stores results, and processes alerts and traps.

To monitor and manage the TCP/IP network at the IBM Thomas J. Watson Research Center, we are currently using the DRAGONS[1] Data Engine, a distributed, object-oriented, run-time environment which supports multithreaded tasks. In this paper, we show how the latencies in polling and alert notification, which can occur in large networks with a central management station, can be reduced by employing multiple Data Engines on multiple hosts to perform management tasks simultaneously. While the DRAGONS Data Engine is a general purpose, run-time environment, it is particularly well-suited to network monitoring, since the problem of polling a large network can be decomposed into small, light-weight queries which map onto the Data Engine's multithreaded environment quite well.

Introduction

The efficient monitoring and management of the components that comprise networks is obviously an important function in any distributed computing environment. In general, if the network fails, the computing environment is rendered useless. As networks have grown in size and importance, network management systems have become critical pieces of software. Unfortunately, network management systems have tended to be single-use, turn-key applications that lack both the flexibility and scalability to deal with situations where the monitored network may include hundreds of routers and thousands of workstations. For such complex networks, there comes a point where it is impractical, if not impossible, to have a central monitoring and data collection point that generates all management queries, stores results, and processes alerts and traps.

One solution to the scaling problem is to distribute the management tasks among different processes in the existing distributed computing environment. This can be attractive if an infrastructure exists that facilitates the development of distributed applications. The DRAGONS Data Engine, developed at IBM Research, provides such an infrastructure and this is the approach we have taken at the IBM T. J. Watson Research Center.

The IBM T. J. Watson Research Center Network

The TCP/IP network at the IBM T.J. Watson Research Center extends to eleven buildings at five sites in Westchester County, New York. The sites are interconnected by either leased T1 lines or fiber. There are over 3500 active TCP/IP systems (AIX, SunOS, DOS, OS/2, VM and MVS) spread across the sites, interconnected by over 20 IP routers. The LAN topology includes token ring, ethernet, and FDDI networks. Besides the local networking infrastructure, links to other sites in North America, South America, Europe and Asia are also monitored.

In addition to the geographically distributed devices comprising the IP networking infrastructure, a large number of local compute- and file servers are monitored and their respective owners are notified (either in real-time or via nightly reports) of outages associated with those systems.

Unfortunately, as more and more routers, workstations and servers were added to the set of devices to be monitored, our existing monitoring methodology did not scale satisfactorily. The

[1]DRAGONS is an acronym for Distributed Reliable Architecture Governing Over Networks & Systems

polling cycle latency (the amount of time it takes to probe and verify the status of each monitored device) was increasing to such an extent that it was becoming infeasible to notify system and network administrators in real-time of failures of important components.

To reduce the polling cycle latency time, we began to use the DRAGONS Data Engine to perform network monitoring and management. It allows us to have a coherent management system that enables the transparent distribution of the workload among a number of systems in the network. However, since our experience with a newly deployed distributed tool like the Data Engine is limited, it was not obvious what was the most efficient workload mix (number of Data Engines and number of threads per host) given our networking infrastructure.

Short Overview of the DRAGONS Data Engine

While the focus of this paper is not on the DRAGONS Data Engine, a short, high-level overview of the environment is useful.

The DRAGONS Data Engine is a distributed, multi-threaded, object-oriented, application development environment. The Data Engine core provides three fundamental abstractions to applications: classes, objects, and threads.

Classes provide a means of organizing data and associated operations in a well-defined fashion. A Data Engine class definition defines the instance variables for each object of a class as well as the methods defined for objects of the class. Methods are similar to functions defined on a specific data type: they are reentrant code bodies that are to be executed as a thread. In the object model defined by the Data Engine, methods are the only external interface to an object. The DRAGONS Data Engine supports multiple inheritance.

An object has several characteristics: it has a unique name (its object ID), it is a member of a class, and it has a state. The state of an object is determined by the values of its instance variables.

The underlying architecture providing the Data Engine run-time environment implements tagged variables that identify the type of variables to the run-time environment. This information is heavily exploited by the free storage reclamation mechanisms and often by applications. Several primitive types are directly supported. These include abstractions for integers, floating points, octet strings, object IDs and native language messages (used to directly support internationalization and locally customizable messages). Composite types such as sparse arrays, associative arrays and sets are also directly supported by the run-time environment. All data can be freely passed between heterogeneous machine architectures.

Method invocations manipulate an object's state. The Data Engine run-time environment creates a new thread for each method invocation. While this may initially appear to be prohibitively expensive, overhead has been kept low and future optimizations may improve performance even more. Using a simple RPC benchmark that is intended to measure overhead of the environment, on an IBM RISC System/6000 Model 560, a Data Engine can process over 2600 method invocations per second and more than 3900 context switches per second. These numbers correspond to over 1300 threads created per second (not all method invocations result in new threads being created because either the method body was null or the invocation was addressed to a thread) and over 650 RPC-style object interactions per second.

Exploiting the Distributed Capabilities of a DRAGONS Data Engine

A basic DRAGONS Data Engine with no local customization includes built-in classes ranging from fundamental, low-level classes that provide access to primitive operating system facilities like timers, files, TCP and UDP sockets to those that implement application support services like interfaces to SNMP agents, a notification service that can display messages on a user's display, send e-mail or drive a pager system, a plotting program to display graphs on a user's terminal or a job scheduler.

DRAGONS Data Engines can be operated as standalone systems; however, by exploiting their support for transparently distributed operation, several interesting possibilities become available. These include utilizing multiple CPUs to process a job, and fault-tolerant operation. The focus of this paper is on exploring the use of multiple CPUs.

The JobController is an example of a Data Engine class that we developed. An object of this class can be instantiated to provide a distributed scheduler function for a job. When the object is created, the maximum number of threads to be created on a particular host for the job is specified, along with the maximum number of hosts that can participate in the computation. If fewer hosts are available than the number specified, the number available is used. By using such an object, an application can be written so that it works when running on a single CPU, but it can automatically take advantage of additional processors when they are made available. The complete implementation of the JobController class, incidentally, is not a complicated piece of code.

The JobController class is used by another sample Data Engine application that polls networks of SNMP hosts. The complete source code for this application appears in Appendix A. A slightly modified version was used to obtain the timing results reported in this paper.

Optimal Distribution of the Polling Workload

While the network management tasks may be distributed over multiple Data Engines, less than optimal performance will be realized unless the scheduling is tuned. This involves determining the number of threads that a given Data Engine should devote to a particular management job, as well as the optimal number of CPUs to be used. Our initial premises were:

- As the latency in the network increases, it is more useful to have increasing numbers of threads that can perform computation while waiting for slow network responses.
- Conversely, as the latency in the network drops, it is more appropriate to have decreasing numbers of threads.
- As the amount of computation required increases, it is more appropriate to decrease the number of threads. As an extreme case, a program that does nothing but computation (no I/O) will run fastest on a dedicated machine which does no timesharing.
- Conversely, if the jobs perform large amounts of I/O, then having more threads allows for the otherwise wasted processor idle time to be used.

With respect to the optimal number of Data Engines, we worked from the following premises:

- A speed-up would only be realized when the unit of work to be distributed was greater than the effort involved in scheduling the work to be done on a remote CPU.
- To achieve as close to a linear speedup as possible for each CPU added to the processor pool, the unit of work to be distributed would have to be constructed in such as fashion as to minimize the amount of interaction with the master object responsible for the overall completion of the job.

Experimental Results in a Local Area Network

To determine the best mix of Data Engines and threads per active Data Engine, a set of experiments was developed to measure polling cycle times while varying the numbers of Data Engines and threads-per-host. This was a simple experiment to setup since the JobController class within the Data Engine performs scheduling of tasks across multiple Data Engines, and varying the number of Data Engines and threads is accomplished without requiring changes or recompilation of the measurement code. We identify the Data Engine running the scheduler as the master Data Engine and any extra Data Engines are called slaves.

As mentioned earlier, the IBM T. J. Watson Research Center complex has IP connectivity to other sites throughout the world, and links to these sites are monitored; however, the target topology measured in the first experiment is a subset intended to be representative of a LAN (vs. a WAN) environment. The subset used for the experiment is an "extended" LAN environment, in which the majority of measured machines are less than three hops away. Only a small fraction of the monitored hosts are reached by a "slow" (T1) serial link. The average round trip time (as measured by ping) between the workstations running the Data Engine and the monitored machines is approximately 15 ms.

A total of 238 hosts were measured, some of which were IP routers with multiple interfaces where each interface on the router was probed on each pass by retrieving relevant MIB variables (see the previously presented source code for details). The majority of the hosts (201 out of the 238) were simply tested by sending an ICMP ECHO request. To poll the 238 hosts, a total of 548 request/response interactions were performed.

The methodology for the experiment was simple: a number of trials were performed, increasing the number of threads per host from one to 30 threads. At each threads-per-host level, trials were performed with an increasing number of processors in the processor pool. Processor pool sizes of one, two and four processors were used. The experimental runs were performed over a number of nights to normalize for bursts of traffic, which in the target environment are less numerous outside of normal working hours, and the average time for each threads-per-Data Engine/processor total combination was plotted. Approximately 100 trials were performed for each data point.

In the best case scenario (four Data Engines each running with seven threads), the polling cycle time was approximately 30 seconds. Given that 238 hosts were polled, this means that the average time need to poll each host in this configuration was 0.13 second. The system was processing a little over 18 request/response interactions per second with the monitored devices. This contrasts to the worst-case performance measured (one thread running on a single Data Engine), in which the average polling cycle time was 104 seconds, yielding a cost of 0.43 second per host. At this rate, the system was processing a little over 5 request/response interactions per second with the monitored devices.

The results of these experiments can be seen in Figure 1. The benefit of multithreading in this application is obvious. A single thread running on a single host takes nearly twice as long to complete the polling cycle as two threads running on a single Data Engine. Obviously, the interesting question is: When does adding Data Engines, and increasing the number of threads per host, stop improving and start degarding polling cycle time performance?

It is apparent that 15 simultaneous threads per host constitute an effective upperbound, after which

performance suffers. The value of this upperbound is due, in part, to the limited number of parameters altered in this experiment. As a consequence, the job scheduler allocates the same amount of work for the master CPU as it does for slave CPUs. This creates contention within the master Data Engine because the dequeued work units compete for timeslices along with the distributed scheduling functions.

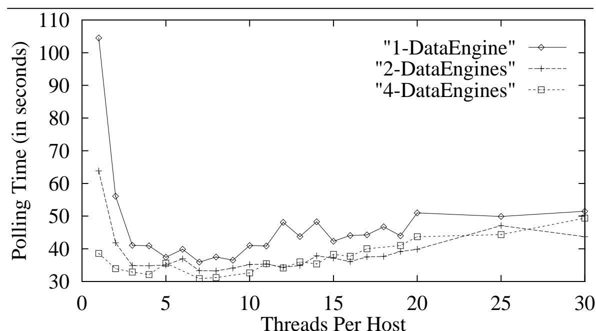


Figure 1: Polling times

In this trial, a single Data Engine does not perform as well as a pool of two or four Data Engines, but the difference between two and four Data Engines is less clear. The graphs illustrate that the actual differences in polling cycle latencies with varying numbers of Data Engines are small, and are only unreasonably large with one or two threads per host. This can explained by the fact that in the target topology, the vast majority of the distributed work units do a trivial amount of work (they merely generate an ICMP ECHO request), and thus the cost of actually sending a task to a remote Data Engine is essentially the same as processing it locally. If the work units delegated to slave CPUs were more computationally-intensive, then we would expect to see larger differences.

To prove this premise, another set of trials was performed using a set of topology data that contained only SNMP-based devices. The results of these trials are shown in Figure 2. The percent reduction in per-host processing time was compared between the original topology and the SNMP-only topology. Using 4 CPUs instead of only one provided a 14% decrease in runtime with the original topology, and a 60% decrease in runtime with the SNMP-only topology. These results support the premise.

In general, it appears that for the original topology of monitored devices, the optimal number of threads-per-processor seems to be between seven and ten. While the use of additional Data Engines does yield better performance, the improvement is not sufficiently large to motivate the use of multiple Data Engines purely for the sake of reducing polling cycle latencies in this particular network. This lack of significant improvement is due to the fact that the topology in question contains an inordinate number of devices which are probed using merely ICMP ECHO requests. When the unit of work to be distributed is more computationally-intensive, the value of using additional Data Engines becomes more apparent.

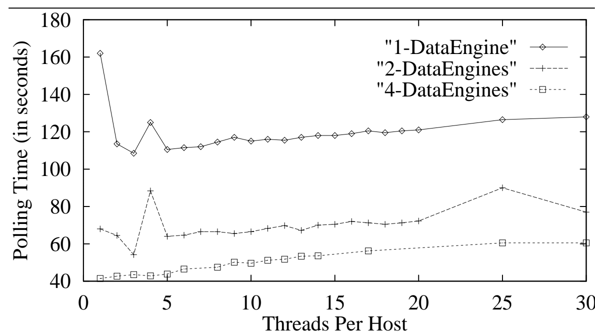


Figure 2: Polling times for SNMP Devices

As noted earlier, performance begins to degrade after about 15 threads per host simply because the master Data Engine becomes CPU-bound. The goal is to create a Data Engine configuration where we have just-in-time monitoring: each Data Engine should run as close to capacity as possible. Multithreading is used to permit a Data Engine to spend time generating queries while waiting for replies to previous queries, and in an ideal configuration, the generation of new queries would end just as replies to previous queries were being returned. The problem exposed by these experimental results is that the master Data Engine becomes a a bottleneck because the distributed scheduler loads the master Data Engine with the same number of work units as the slaves. Better performance in the multi-user case might be achieved by increasing the number of work units offloaded to the slaves and decreasing the number of work units assigned to the master Data Engine.

To gain some insight into the effect of the default equal bias-scheduling policy, a third set of trials was performed with the scheduler instructed to avoid delegating any work units to the master Data Engine. In such a scenario, the master Data Engine is responsible only for scheduling and the slaves are responsible for actually making progress on the job. The results obtained demonstrated that the mechanics of distributing work units among multiple hosts becomes the bottleneck.

A consistent lower bound was observed, regardless of the number of hosts made available, indicating that the master Data Engine was 100% saturated and the slave CPUs were processing work as quickly as it could be delegated. As a result, we have determined that to further improve performance, we will need to invest more effort in the implementation of the JobController task. Two new approaches present themselves. One is to attempt to reduce the number of method invocations per monitored device that must cross host boundaries. The current total is four

and it should be straightforward to reduce this to two method invocations. The second approach, that is not mutually exclusive with the first, is to attempt to distribute the scheduling function itself. This is not as straightforward as a static (pre-calculated) distribution of work units that does not take into account effects introduced by lost packets and unresponsive hosts.

Experimental Results in a Wide Area Network

The results obtained from a LAN environment with low network latency were in line with our expectations, but the effect of multiple DRAGONS Data Engines was not dramatic enough to compel their use. We had run trials to confirm that multiple Data Engines had a greater effect when the distributed work units were more computationally-intensive, but we felt it would be useful to explore the performance of Data Engines in a wide area (T1 and T3 based) TCP/IP network backbone.

This set of trials was conducted on the ANSnet backbone network. The characteristics of this network are different from the Watson LAN environment since it is composed primarily of serial links. In addition, the monitored devices are generally further away in terms of numbers of hops. As the number of hops increases, a corresponding increase in round-trip times is measurable. The average round trip time between the measurement points and monitored hosts in this network is 52 ms, almost 3.5 times longer than in the Watson IP LAN.

The same code and experimental methodology was used for this target environment with two modifications: a maximum of two Data Engines were used, thus eliminating the case of four processors in the processor pool which was tested against the Watson Research Center LAN; and the processors were slower. The results are shown in Figure 3.

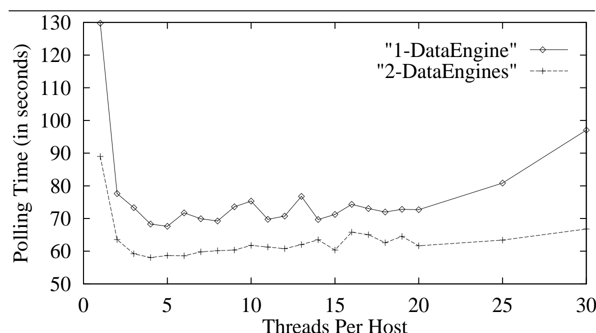


Figure 3: Polling times with two engines

In this environment, the addition of a second Data Engine had an obvious effect on the polling cycle time. Between 10 and 20 threads, the second Data Engine reduced the polling cycle time by 10 seconds. In the best case configuration (two Data Engines each running six threads) the average polling cycle time was about 60 seconds. For a total of 98 monitored hosts, the average polling time per host was 0.61 second. This contrasted with the worst case scenario (one thread running on one host) where the average polling cycle time was 129 seconds, or 1.31 seconds per host.

In both the one and two Data Engine cases, performance began to degrade noticeably at about 20 threads per host. Again, the bottleneck of the distribution of work units limited throughput.

Comparing the LAN and WAN results

In the ANSnet backbone, the probability of packet loss is greater than that of the Watson LAN. We see the benefit of using more than one Data Engine to compensate for the loss of throughput when a thread is held up waiting for a retransmission to take place. In the single processor case, a packet loss event temporarily drops the system back to making progress with T - 1 threads, whereas in the two processor case 2 * T - 1 threads remain active.

Better performance is achieved with a fewer number of threads in the case of the ANSnet backbone as contrasted with the Watson LAN. This is as expected: there are no trivial units of work being distributed because all of the hosts being polled are SNMP-based. Since the computation involved in dumping the tables of a router is greater than in generating an ICMP ECHO request, fewer units of work are required to saturate a given processor.

Summary and Future Directions

- The use of multi-threading is a very effective technique for reducing polling latencies.
- The use of multiple Data Engines also contributes to a reduction in polling latencies.
- Multiple Data Engines are more effective when the unit of work to be distributed is computationally-intensive; otherwise, the effort involved in delegating the work to a remote processor far outweighs the savings gained by offloading the work.
- Our simplistic distributed scheduling application has become a bottleneck for increased performance.

The issue of efficient workload distribution needs to be addressed with several avenues of exploration not discussed in this paper.

- The mechanism that starts and controls work units on remote hosts can be improved. This currently involves 4 method invocations which must cross host boundaries.
- An alternative approach to the current dynamic scheduler is to partition the topology database into fixed set of hosts and delegate these to slave CPUs. The slave CPUs would send one message back to the master informing it of the status information they had collected. On the negative side, such a static

scheduling approach makes it more difficult to add or remove CPUs from the processor pool while the job is in progress.

- Data Engines already understand the concept of delegating requests for certain networks or hosts to authoritative Data Engines. This is used, for example, to automatically route requests for external networks through secure IP gateways, enabling transparent SNMP access access to external networks. It can also be used in a wide area network environment to route processing of requests closer to their ultimate destination, thus eliminating the latency involved in repeated interactions with a very remote site.

Availability

DRAGONS software can be licensed for the IBM RISC System/6000 and Sun (SPARC-based) platforms. Inquiries can be sent to dragons@watson.ibm.com or the authors.

Author Information

Robert Lehman is a Staff Programmer in the Watson Networking Systems group at IBM Research. He is responsible for management and monitoring tools for the Watson IP network. He can be reached ~~via US Mail at IBM T. J. Watson Research Center, PO Box 704, Yorktown Heights, NY 10598, and~~ via networked electronic mail at ~~rlehman@watson.ibm.com.~~

Geoffrey C. Carpenter is an Advisory Programmer at IBM Research. He works in the Advanced Information Technology Group (Department of Computing Systems) and is the author of XGMON, an SNMP manager for TCP/IP networks, and is the key developer of DRAGONS. He can be reached ~~via US Mail at IBM T. J. Watson Research Center, PO Box 218, Yorktown Heights, NY 10598, and~~ via electronic mail at ~~gcc@watson.ibm.com.~~ gcc@fargos.net

Nguyen C. Hien is with IBM Research, presently the Manager of Automation Systems in the Advanced Information Technology Group (Department of Computing Systems). He has development responsibility for Systems and Network Management tools, in particular XGMON, an SNMP manager for TCP/IP networks, and DRAGONS, an object-oriented environment for distributed applications. He can be reached ~~via US Mail at IBM T. J. Watson Research Center, PO Box 218, Yorktown Heights, NY 10598, and~~ via electronic mail at ~~hien@watson.ibm.com.~~ hien@fargos.net

---

Appendix A: Polling Code

```
include "DEinterfaces.oog_h"
include "DEtype_util.oog_h"

enum states {
        IN_PROGRESS, COMPLETED, NO_RESPONSE, NO_SNMP_RESPONSE
};

class SNMP_PollNetwork {
        oid             controller;
        int             startTime;
        int             jobsCreated;
        int             jobsDone;
        int             tc, hc;
        array(string)   community;
        array(int)      addresses;
        set(oid)        hosts;
} inherits from Object;

SNMP_PollNetwork:create(int threadCount, int hostCount)
{
        int             t, h;

        t = 10; h = 1;

        if (argc >= 1) t = threadCount;
        if (argc == 2) h = hostCount;

        tc = t; hc = h;

        send "schedule"(tc, hc) to thisObject from nil;
}

SNMP_PollNetwork:schedule(int threadCount, int hostCount)
{
```

```
        int             i;
        oid             host;

        write_cout("Starting to schedule, t = ", tc, " h = ", hc, "\n");

        if (controller == nil)
                controller = send "createObject"("JobController", tc, hc)
                                to RootObject;
        if (hosts == nil)
                hosts = send "allInstances"("SNMPhostInfo") to RootObject;

        startTime = localRelativeTime();

        i = 0;
        for host in hosts {
                i = i + 1;
                if (community[i] == nil)
                        community[i] = send "getReadCommunity" to host;
                if (addresses[i] == nil)
                        addresses[i] = send "getAddresses" to host;
                send "queueJob"("SNMP_PollHost", thisObject, host,
                        community[i], addresses[i]) to controller;
                jobsCreated += 1;
        }
}

SNMP_PollNetwork:delete()
{
        send "doneWithJob" to controller;
}

SNMP_PollNetwork:doneWithHost(oid obj, int success)
{
        int             endTime;

        jobsDone += 1;
        if (jobsDone == jobsCreated) {  // all done...

                endTime = localRelativeTime();
                write_cout("All ", jobsDone, " hosts polled in ",
                        endTime - startTime, " seconds!\n");
                jobsDone = 0;
                jobsCreated = 0;
                send "schedule"(tc, hc) to thisObject from nil;
        }
}

// poll a host
class SNMP_PollHost {
} inherits from Object;

// Some constants here for performance rather than looking them up...
const           sysUpTimeObjID = "1.3.6.1.2.1.1.3.0";
const           ifNumberObjID = "1.3.6.1.2.1.2.1.0";
const           ifTypeObjID = "1.3.6.1.2.1.2.2.1.3.";
const           ifOperStatusObjID = "1.3.6.1.2.1.2.2.1.8.";
const           ifInUcastPktsObjID = "1.3.6.1.2.1.2.2.1.11.";
const           ipAdEntAddrObjID = "1.3.6.1.2.1.4.20.1.1";
const           ipAdEntIfIndexObjID = "1.3.6.1.2.1.4.20.1.2";
const           ipAdEntNetMaskObjID = "1.3.6.1.2.1.4.20.1.3";
const           ipAdEntBcastAddrObjID = "1.3.6.1.2.1.4.20.1.4";
const           ipRouteIfIndexObjID = "1.3.6.1.2.1.4.21.1.2";
const           ipRouteNextHopObjID = "1.3.6.1.2.1.4.21.1.7";
const           ipNetToMediaNetAddressObjID = "1.3.6.1.2.1.4.22.1.3";
```

```
const           MAX_VARS_PER_REQUEST = 5;

SNMP_PollHost:create(oid master, oid hostInfo, string community,
        array(any) addresses)
{
        string          hostAddress;
        string          objID, ifNum;
        int             i, addr, end_block, count;
        oid             mibDirectory, sqeInterface;
        int             ok, rtt;

        set(oid)        obj_set;
        set(any)        result_set;
        array(any)      result;
        array(int)      addrTable;
        array(int)      addrIndex;
        int             addrCount;
        string          haltKey;
        int             haltKeyLength;

        obj_set = send "allInstances"("SNMP_QueryEngineInterface")
                        to RootObject;

        for sqeInterface in obj_set break;
        if (sqeInterface == nil) {
                write_cout("No SNMP Query Engine Interface on this host\n");
                exit;
        }
        obj_set = send "allInstances"("SNMP_MIB_Directory") to RootObject;

        for mibDirectory in obj_set break;
        if (mibDirectory == nil) {
                write_cout("No SNMP Query Engine Interface on this host\n");
                exit;
        }
        //
        // First, find interface that responds to SNMP
        //
        i = 0;
        addr = addresses[i];
        ok = 1;
        while (ok) {
                hostAddress = dottedAddress(addr);
                if (community == "PINGONLY") {
                        rtt = send "pingHost"(addr) to sqeInterface;
                        if (rtt != -1) {        // no SNMP response, but up
                                send "doneWithHost"(hostInfo, COMPLETED)
                                        to master;
                        } else {
                                send "doneWithHost"(hostInfo, NO_RESPONSE)
                                        to master;
                        }
                        exit;
                }
                result_set = send "getMIBvalue"(addr, community,
                        sysUpTimeObjID) to sqeInterface;

                if (result_set != nil) ok = 0;
                if (ok == 1) {  // no SNMP response...
                        rtt = send "pingHost"(addr) to sqeInterface;
                        if (rtt != -1) {        // no SNMP response, but up
                                send "doneWithHost"(hostInfo, NO_SNMP_RESPONSE)
                                        to master;
```

```
                                exit;
                        }
                        i += 1;
                        addr = addresses[i];
                        if (addr == nil) ok = 0;
                }
        }
        if (addr == nil) {      // complete failure!
                send "doneWithHost"(hostInfo, NO_RESPONSE) to master;
                exit;
        }
        hostAddress = dottedAddress(addr);
        //
        // Second, dump IP address table for ifIndex->IP address
        // mapping for this host.
        //
        obj_set = emptySet;
        obj_set += ipAdEntAddrObjID;
        obj_set += ipAdEntIfIndexObjID;
        ok = 1;
        haltKey = ipAdEntAddrObjID + ".";
        haltKeyLength = length(haltKey);
        while (ok) {
                result_set = send "getNextMIBvalue"(addr, community,
                        obj_set) to sqeInterface;
                if (result_set == nil) {
                        rtt = send "pingHost"(addr) to sqeInterface;
                        if (rtt != -1) {        // no SNMP response, but
                                                // up
                                send "doneWithHost"(hostInfo, NO_SNMP_RESPONSE)
                                        to master;
                                exit;
                        }
                        send "doneWithHost"(hostInfo, NO_RESPONSE) to master;
                        exit;
                }
                i = 0;
                obj_set = emptySet;
                ok = 0;         // only successfully query disproves
                                // this...
                for result in result_set {
                        objID = result[0];
                        obj_set += objID;
                        if (i == 0) {   // ipAdEntAddr
                                objID = midstr(objID, 0, haltKeyLength);
                                if (objID == haltKey) { // still in table
                                        ok = 1;
                                        addrTable[addrCount] = result[2];
                                }
                        } else if (i == 1) {    // ipAdEntIfIndex
                                if (ok == 1) {  // still in table
                                        addrIndex[addrCount] = result[2];
                                }
                        }
                        i += 1;
                }
                if (ok == 1) addrCount += 1;
        }
        //
```

```
        // Third, dump interface table:  get interface status and
        // packet count.
        //
        count = 0;
        while (count < addrCount) {
                end_block = count + MAX_VARS_PER_REQUEST;
                if (end_block > addrCount) end_block = addrCount;
                obj_set = emptySet;
                // ask for interface type, status for each interface
                for (i = count; i < end_block; i += 1) {
                        ifNum = to_string(addrIndex[i]);
                        obj_set += ifOperStatusObjID + ifNum;
                        obj_set += ifInUcastPktsObjID + ifNum;
                }
                // we don't actually care what the results are
                // here...
                send "getMIBvalue"(addr, community,
                        obj_set) to sqeInterface from nil;
                count = end_block;      // next block of
                                        // MAX_VARS_PER_REQUEST
        }
        send "doneWithHost"(hostInfo, COMPLETED) to master;
}

SNMP_PollHost:delete()
{
}
```