A UNIX Implementation of HEMS

Craig Partridge
NSF Network Service Center (NNSC)
at BBN Laboratories Incorporated†

Abstract

The High-Level Entity Management System (HEMS) is currently the best known of the network management schemes designed to work on TCP-IP networks. In this paper, the author describes experience with the first HEMS implementation, done under 4.3BSD.

1. Introduction

In late 1986, a number of researchers, users and vendors came to the conclusion that the TCP-IP protocol suite needed a standard internetwork management protocol and that none of the then existing management protocols was a suitable candidate for standardization. Working groups were formed to try to develop new management systems and protocols which could be considered for standardization. The High-Level Entity Management System (HEMS) is the best known of the systems to come out of this effort.

This is a detailed description of the first HEMS implementation. The goals of this implementation were (1) to examine HEMS from a programmer's standpoint (instead of a system designer's); (2) to confirm that HEMS could be implemented; and (3) to acquire some performance information. The 4.3BSD system was chosen as the host environment because the BSD system was well-known. HEMS.

The presentation is broken up into three parts, corresponding with the goals of the implementation. The first section illustrates how one might use HEMS. (Readers interested in a detailed specification or a study of motivation are encouraged to read the references [1,2]). The second section describes the architecture of the implementation, and can probably be skipped by non-implementers. The third major section describes HEMS performance. Finally, in the conclusion, I try to summarize the current state of HEMS.

2. Overview of HEMS

HEMS is designed to provide the most critical network management function: a system-independent mechanism for performing monitoring and control of remote internetwork nodes. To put that another way, HEMS makes it possible to observe and manipulate remote nodes on an IP network. Such a capability is essential if effective internetwork management tools are to be developed. But HEMS does not deal with how to build the tools themselves. One can think of it as analogous to the *ptrace* system call; something like *ptrace* was required to allow us to write debuggers such as *sdb*, *adb* and *dbx*

In HEMS, every IP node in a network (e.g., a host, terminal server or gateway) supports a hierarchical database which presents an abstract interface to the internals of the node. To perform management operations, applications send database queries to a query processor, or *agent*, on the node to be managed. The query language is designed to be inexpensive to process. Query language operations which write into the database are mapped into operations which change the internal state of the

---

† The author's mailing address is: c/o BBN Laboratories Inc., 10 Moulton St, Cambridge MA 02238. His electronic mail address is craig@nnsc.nsf.net

node. So for example, changing the section of the database which represents the system routing table causes changes in the actual system routing table. Operations which read the database are mapped into operations which extract information from the node. Queries and the replies to queries are formated in the self-describing data format, Abstract Syntax Notation 1 (ASN.1).

The meanings of query language operations and values in the database are standardized. The same database item retrieved from two nodes will have the same meaning and format and the values will be comparable, even if the nodes are made by different manufacturers. For example, the portion of the abstract database corresponding to the routing table looks the same on every machine, regardless of how the actual routing table in the node is implemented. The goal is to create a system where a network manager trying to diagnose a problem can send queries to the node without having to know anything about the node other than its IP address.

HEMS can be used to manage all levels of the network stack in a node from the interface layer up to applications such as *telnet* and *rlogin*. Proxy and translator agents can be used to manage network components such as bridges, repeaters and modems, which may not have IP addresses. Proxy agents are agents which reside on a node with an IP address, which are designated as responsible for a component. A good example of a proxy is an IP node to which the control line of a modem is attached. The IP node is the only node which can effect control operations on the modem. Translators have a non-exclusive responsibility for a node. For example, a bridge might be managed through any one of several IP nodes, each of which is capable of controlling the bridge by translating HEMS requests into some link-level management protocol.

3. Details of HEMS: Four Examples

This section presents four examples, designed to illustrate how HEMS can be used.

3.1. A Small Database

For simplicity, these examples assume that we are attempting to remotely manage a network node that supports the simple abstract database shown in Figure #1. (The actual database proposed for use on IP nodes is much more complex and contains over 200 separate items).

The database contains three items: a status register, the routing table (which contains the individual routing entries) and a routing distance indicator.

The status register is a virtual hardware status register. In the examples we use it to allow us to reboot the node by writing the value 0 into it.

The routing table is a set of individual routing entries, and corresponds to the actual routing table used by the node. Each routing entry contains three items of information: the destination network or host (dest), the next hop gateway (next), and the distance to the destination using some metric (metric).

The routing distance indicator tells us how the node measures distance to a network. For the purposes of this example, we assume that two different distance measurement techniques are possible: a delay-based scheme (i.e., a network is so many milliseconds away), and a hop-count scheme (a packet must traverse a certain number of gateways to reach the network). If the delay-based scheme is in use, then the routing indicator has the value 1, and the distance metric in the routing entries is interpreted as a time value. If the hop-count scheme is in use, the routing indicator is set to 2, and the distance metric is the number of hops.

One important point about the database is its tree-shaped structure. The HEMS query language requires that any database it manipulates be a tree. This database schema was chosen because it is faster and easier to manipulate than a relational schema, and we were interested in reducing processing costs at the node.

3.2. Abstract Syntax Notation 1 (ASN.1)

Because HEMS requires that data from any node be intelligible to any other node, it needed to use an external data formats. External data formats are standardized data formats that all nodes use to exchange information; if a node's data formats are not the same as the external format, it must convert its data into external form before sending it to any other node. For HEMS, we chose to use the OSI

external data format, Abstract Syntax Notation 1 (ASN.1) [3,4].

ASN.1 is a ''tagged'' data format. Each item of data is a triple of the form <tag,length,data>. The tag is a number that encodes information about the data (its type, whether it is a structure, etc.). The length is the length of the data section. There is no limit on the size of the length or tag fields. The fields are both self-describing and can contain arbitrarily large numbers. The data is either a value (e.g., an integer) or a collection of ASN.1 objects. Because ASN.1 objects can contain other ASN.1 objects, the data format can support extremely complex data structures including unordered collections of data, in which each item is identified by its unique tag.

Another advantage of tagged data is that the sender and receiver do not have to agree in advance on the format of the data. Untagged data formats require the receiver to ''know'' the datatypes the sender is transmitting (for example, two integers followed by a character string). That was too restrictive for the query language envisioned for HEMS. We wanted to use query language operations with variable numbers of operands, and needed tagged types to identify each operand.

Figure #1 includes the ASN.1 type for all data items and some of the examples below show the binary ASN.1 encoding of the values. But for readability the examples are primarily written in a symbolic notation. In this notation, a name, such as ''RoutingEntry'', represents an ASN.1 data tag with no data; a name followed by a value in parenthesis is an ASN.1 object with a value (for example, ''metric(10)'', is a metric field with a value of 10); and a name followed by brackets such as ''RoutingEntry{metric,next}'' is a nested type, with individual fields listed within the brackets.

Query #1	
Symbolic	**Hexidecimal Encoding**
RoutingTable GET	8100410101

Query #2	
Symbolic	**Hexidecimal Encoding**
RoutingTable BEGIN	8100410102
RoutingEntry Filter{item{equality{dest(192.5.58.0)}}} GET	80008006800380c0053a410101
END	410103

Reply #1			
Symbolic			**Hexidecimal Encoding**
RoutingTable{	8112		
	RoutingEntry{	81000f	
			dest(192.5.58.0),next(128.89.0.92),metric(2)
			}
}			

Reply #2			
Symbolic			**Hexidecimal Encoding**
RoutingTable{	8124		
	RoutingEntry{	81000f	
	RoutingEntry{		81000f
		dest(192.5.58.0),next(10.4.0.5),metric(1)}	8003c0053a81040a040005820101
}			

Query #3	
Symbolic	**Hexidecimal Encoding**
StatusRegister(0) SET	800100410108

Query #4	
Symbolic	**Hexidecimal Encoding**
RoutingTable GET	8100410101
RoutingIndicator(2) SET	820102410108

3.3. Example 1: Finding A Particular Route.

Assume that we are trying to send data through a gateway to a remote network (192.5.58.0), and for some reason, the data does not seem to be getting through. After confirming that our machine is sending the packets to the gateway properly, we would like to check the routing tables in the gateway.

In HEMS, this is done by sending a HEMP query message to the gateway. HEMP is the HEMS message protocol. A *query* message, is a particular type of HEMP message, which asks for information. All HEMP messages come in two parts, a standard header, which contains some basic information about the message (whether the body of the message is encrypted, what access permissions it has, an ID number, the message type, etc.), and the body of the message, which is a sequence of ASN.1 objects.

The interpretation of the body is per-message-type specific.

For queries, the body of the message is an ASN.1-encoded database query, written in the HEMS query language. The language is a sequence of operands punctuated by operations on those operands (i.e., operations follow their operands). An operation may have a variable number of operands. The operations can retrieve data, change data, delete or add data, or manipulate the context stack that is used to keep track of what part of the database is being used.

Looking at our example, what we want to do is retrieve a route, so we would use the GET instruction to extract the route from the database. There are several ways to do this. The simplest request is shown in Query #1, which extracts all the RoutingEntries which make up the routing table and returns them to our application. We then would have to read the entire table to find the particular route of interest. This is a rather expensive approach; if the node is of any importance, its routing table is likely to contain a few hundred entries. The cost of dumping all those entries onto the network and then processing them at our end to find a single route is clearly excessive.

To allow a remote user to retrieve selected items of information, HEMS supports *filters*, a method for selecting items using boolean expressions. In our example, we wanted the RoutingEntry which has a destination network of 192.5.58.0, so we can use a filter on the *dest* field, as shown in Query #2.

In this case, the GET command takes two arguments, the data to retrieve (a complex RoutingEntry) and the filter that tells us which RoutingEntry is the one we want. Note that we've also introduced two other operations, the BEGIN and END operations. Filters only work on sub-nodes of the current node of the tree, so we have to move ourselves into the node from which we want to do the filter. BEGIN is the HEMS version of *pushd*, and moves us to a new context by manipulating the context stack; END moves us back to our previous context by popping the context stack.

Query #2 is probably the one an application would use, although fancier versions could be imagined. So what comes back?

When a node processes a HEMS query, it sends back a HEMP *reply* message. The body of the reply message is the answer to the query. In the case of Query #2, if the route existed and went through gateway 128.89.0.92, the reply might be the one shown in Reply #1. If two routes existed, the reply would be Reply #2.

If no route existed, the reply would simply be an empty RoutingTable structure. Note a nice feature of these replies. They represent walks of the database tree. In fact, all HEMS replies are a walk of the database tree (possibly with multiple visits to the same node).

Note that the form of the replies is the same regardless of what type of node is queried. The node is required to convert from its internal format, such as the 4.3BSD *rtentry* struct, into the ASN.1 structure used by HEMS.

3.4. Example 2: Rebooting The Machine.

Example #1 showed how one can do monitoring with HEMS. In this example we look at the more complex problem of control (being able to change how a node behaves).

Imagine that we want to reboot a remote machine. The HEMS query to tell a machine to reboot itself is simple. We need only load the value 0 into the status register, using the SET operation, as in Query #3. When a node receives this query, it is expected to take whatever action is required to cause itself to reboot. For example, on a BSD host, the HEMS agent would invoke the *reboot()* system call. On a machine where the agent ran in privileged mode, it might just write a bit in a status word.

The complexity of control operations is not in the operations themselves; the SET, ADD and DELETE operations are like all other operations. The problem is what the operations allow a remote user to do. It is one thing to allow a remote user to retrieve monitoring data; it is quite another to allow remote user to change how a box behaves. Support for control operations all but requires support of access control and encryption mechanisms.

In HEMS, these facilities are provided by HEMP. At the start of each HEMP message are optional sections for authentication and encryption information. The authentication section allows remote users to prove their membership in one or more access groups. We preferred per-message authentication to

per-operation authentication because we felt is was more efficient to processing authentication information once at the start of a message. The encryption sections allow remote users to indicate what encryption method they are using in their message, and what type of encryption they would like in the reply.

3.5. Example 3: Learning That The Machine Is About to Reboot

There is something left out of Example #2. What happens if we are not the only people watching the node we reboot? How do we notify everyone that this reboot is planned and not the sign of hardware or software problems?

It turns out that the best way to notify everyone is to have the node tell them. When a node receives instructions to reboot, it sends out a HEMP *event* message, which notifies interested managers that the machine is about to reboot. Managers that care about whether a reboot is planned can deposit their address with the HEMS agent. Such requests to be notified are on a per-event basis; a manager can select the events it wants to receive.

Like everything else in HEMS, events are standardized. The same reboot message is sent from every machine when it is about to reboot. One of the outstanding pieces of work left for HEMS is defining all the possible events. Research indicates that there are probably about 500 to 1,000 distinct events that need to be defined, a massive task.

3.6. Example 4: Changing The Routing Configuration.

Finally, we look at a more complex HEMS scenario. Imagine that we have misconfigured a node to use delay-based routing metrics, when all other nodes in our network use hop-count metrics. As a result, our node is distributing misleading routing information to the other nodes, and is misinterpreting the routing information it receives. We would like to fix the routing metric, and also look at the node's routing table to see how badly corrupted it is. A query to do this is shown in Query #4.

This query extracts a copy of the routing table and then sets the routing indicator to use hop-count metrics. Because the routing indicator is a key variable, the node would probably send an event message when it is changed. Notice that HEMS allows us to mix monitoring operations with control operations in a single message. This is one of the unique features of HEMS. The other major management protocols do not support this feature [2].

4. The Implementation

This section is a detailed look at the implementation of the HEMS agent under 4.3BSD. The purpose is to illustrate one way an agent might be implemented. A question sometimes raised by those who read the HEMS specifications is whether it is too complex to reliably implement. I believe this implementation shows that the HEMS is not extraordinary in its complexity.

To limit the complexity of the system, the implementation breaks the agent down into four distinct modules: an ASN.1 preprocessor, that converts inbound and outbound streams of ASN.1 data into a convenient internal form; a HEMP message processor, that reads and generates HEMP messages; a query processor which reads and interprets the query language sent in queries; and a complex database structure which allows the query processor to map between ASN.1 objects and kernel data structures. A diagram of relationships between each module is shown in Figure #1. We look at each module in detail below.

4.1. HEMP Message Processor

Whenever a HEMP query is received over the network, the new query is passed to the message processor. The message processor reads the header of the message, which involves checking version numbers, processing any authentication mechanisms, and, if necessary, enabling encryption and decryption. Finally, it generates the header of the reply message and calls the query processor to handle the body of the query.

4.2. Query Processor and Abstract Database

The query processor does most of the work in the HEMS agent. At its heart is a short loop:

```
WHILE there are more ASN.1 objects
    READ the next ASN.1 object
    IF the object is an operation
        do the operation
    ELSE the object is an operand and
        save it for the next operation.
```

The real complexity in the processor is in performing the operation.

The perform an operation the agent needs to solve two problems. First it needs to properly interpret the operands. Second, it must effect the operation on the database and the host node.

Interpreting the operands was the harder problem. Every operand must be interpreted in the current context in the database tree because the ASN.1 tags which identify the values are context-specific. Furthermore, the operands usually can be arbitrary complex. For example, we can write requests which change some values of a routing entry, but not others. For example, Query #5

Query #5

Symbolic	**Hexidecimal Encoding**
RoutingTable BEGIN	8100410102
RoutingEntry{metric(1),next(128.89.0.1)}	8009820101810480590001
Filter{item{equality{dest(192.5.58.0)}}} SET	8006800380c0053a410108
END	410103

instructs the agent to change two of the fields in the selected RoutingEntry and leave the other fields untouched. There is a tricky data structure problem here. We need to be able to combine information from the context stack, the database and the operands to determine where the operation is to be applied.

The implementation solves this problem by using the same data structure for the database, the operands and the context stack. All three are represented by a tree structure, which mimics the nesting of the ASN.1 objects. So the RoutingEntry operand of the last example, would be represented as a tree of depth two, with the root representing the RoutingEntry, and having two children, one for the metric field and one for the next hop field. The context stack is a pointer into the tree structure which implements the HEMS database. The implementation then uses a set of routines which can use one structure as the template for applying an operation to another structure. Again, looking at the example, once the filter has located the proper RoutingEntry to change, the agent calls a routine with the operand containing the fields to be changed, and the RoutingEntry we wanted to change, and the routine makes sure that the SET operation is applied to the metric and next hop fields of the old RoutingEntry. Note that support of filters is also made easier by this method. The tree structure of the data makes a good parse tree to hand to an expression handler; filters do not have to be further processed before they are evaluated.

This brings up the second problem: what does it mean to do a SET on an object? (Or a GET, ADD, or DELETE for that matter). Because the database is an abstraction of the host node, reading from it or changing it is not a matter of simply changing the database. The agent must change the state of the node to correspond to changes in the database.

There are at least three approaches: (1) One can maintain a full database (possibly on disk) and run a background daemon that keeps the database and the actual host software consistent. The major problems with this approach are that if the daemon fails, the remote manager may not see this because the database will still be accessible, and that running such a daemon is expensive given that one expects only a subset of the database to be accessed on a regular basis. (2) One can maintain a full database but integrate it tightly into the host software. So, for example, the code which maintains the routing section of the database is also the same code that accepts routing updates from protocols and chooses routes for datagrams. This approach makes a lot of sense but requires a major rewriting of the host

software. Or, (3) one can maintain a database of entry points into the host software. Except for a few static values, the database contains no real data, just methods for manipulating the host software based on remote requests. So for example, on the BSD system, when a remote request to read the routing table is received, the database reads */dev/kmem*, and repackages the BSD routing structures into ASN.1. A request to change the routing table is translated into a sequence of *ioctl()* calls. This is the approach used in the implementation. It works reasonably well, except for the nuisance of having to write special access routines for portions of the database.

4.3. ASN.1 Preprocessor

One of the two hardest problems in the implementation was finding a way to limit the amount of input processing that needed to be done. The agent reads from a stream of ASN.1 objects. This stream may be encrypted, and it may be convenient to buffer the data. Furthermore, ASN.1 uses variable length fields, so processing the objects in their ASN.1 format more than once is undesirable.

One option is to try to fully process each object as it is received. This approach doesn't work well in HEMS. Query language operands really need to be processed twice; once upon receipt, when they are scanned and stored waiting for the operation, and again when they are actually used by the operation. Filters are accessed repeatedly during processing. So the implementation uses another approach, that of converting each object as it is received into the internal format described above. The ASN.1 preprocessor is responsible for doing the conversions to and from ASN.1. It also quietly does encryption and decryption of data streams.

Modules which are reading data, call on the routine *asn_read()*, which returns the tree-shaped structure which is the internal representation of the next ASN.1 object in the input stream. This internal representation incorporates all ASN.1 tags and length information, but maintains it in an easy-to-access format. Figure #2 shows how a sample routing entry might be represented in this form.

Modules which are writing data call on the routine *asn_write()* which takes the tree-shaped internal representation and generates and sends the ASN.1 representation.

Both *asn_read()* and *asn_write()* quietly do data encryption and I/O buffering. Another routine, *asn_flush()* is invoked at the end of all processing to force transmission of any partially buffered or encrypted data.

5. Performance

One of the key goals of HEMS was to design a system that did not overburden the machines it ran on. The view we took when designing HEMS was that it should be possible to develop a full function internetwork management protocol, without consuming huge amounts of memory space or CPU processing. We wanted it to be possible to run HEMS on the a 68000-based IP routers that have recently become popular.

To gather information on whether HEMS met these goals, several tests were performed with this implementation on a SUN 3/50 workstation. These tests were not intended to be definitive, because the implementation is not highly optimized. The goal was to gather some information which would tell us whether HEMS was likely to meet its goals. This section discusses the results of the tests and what they tell us about HEMS performance.

5.1. Code Size

Currently the HEMS agent has a running image size of about 83Kbytes (this after running a few queries through it). That is a little bit bigger than we had hoped. When we were originally designing HEMS, an unofficial goal was an image size of no more than about 100Kbytes. Since the implementation is incomplete (in particular, event handling and some parts of the database are not yet supported) it is not clear that this particular implementation will meet that goal. At the same time, since the implementation is nearly complete, and could presumably be optimized to use less data, we probably won't miss 100Kbytes by too much.

5.2. Processing Costs

Two sets of tests were run to measure the cost of processing HEMP query messages. Both tests used the *gprof* profiler to get performance times.

The both tests tried to estimate the cost of receiving and processing a query. Two sample queries were developed. A short query which retrieved two values (a BEGIN, two GETs and an END instruction) and a medium-length query which extracted the same information plus the entire routing table (which contained 10 routes). Each query was sent to the agent one thousand times per profiling run. The profiles were then analyzed to determine cost of processing the queries, and also to estimate which modules of the implementation were more expensive.

The per-query costs are shown in Table #1. Note that these numbers are not terribly precise. *Gprof* showed considerable variation (about 15-25%) in processing times from one test to another. As one indication, the HEMP header was the same for both queries, so the HEMP processing time should have been the same.

Table #1
(in milliseconds)

Query	HEMP Processing	Query Language Processing	Total Processing
Small	8.1	19.8	27.9
Medium	6.1	41.2	47.3

Those numbers are pretty good. Current experience with Internet gateways suggests that the average gateway receives a query once every few minutes, and in peak traffic periods might get as many as two or three a second. Taking the numbers for medium size query, this suggests that at peak load of three medium sized packets per second, about 15% of the gateway's processing time would be consumed with management requests.

Note that the 15% figure makes a lot of assumptions. Gateways often use tasking operating systems which are very sensitive to context switches. HEMS is likely to require at least two context switches per-packet, one each in and out of the agent task. Furthermore, the processing costs appear to be highly sensitive to the number of ASN.1 objects in the query.

The *gprof* output was also screened to try to determine where the agent spent most of its processing time, by module of the software. This information is shown in Table #2.

Table #2
(in percent)

ASN.1 Preprocessor	Query Processor	HEMP Processor	Misc
62%-75%	6%-32%	1%-2%	5%-17%

The table suggests that the majority of the processing costs are in the ASN.1 preprocessor, reading and writing ASN.1 objects. Careful reading of the profile suggests the majority of costs are spent in UNIX system calls (e.g., *read* and *write*) not in the ASN.1 parser.

6. Conclusions

Bibliography

[1] C. Partridge and G. Trewitt, The High-Level Entity Management System (HEMS); RFCs 1021-1024. In *Network Working Group Request for Coments, no. 1021-1024*, Network Information Group (NIC), SRI International, Menlo Park, Calif., Oct. 1987.

[2] *IEEE Network*, Vol 2, no. 2. Special Issue on Internetwork Management Protocols. March 1988.

[3] *Information processing systems - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1).* International Standard, no. 8824. International Organization for Standards, May 1987.

[4] *Information processing systems - Open Systems Interconnection - Basic Encoding Rules for Abstract Syntax Notation One (ASN.1).* International Standard, no. 8825. International Organization for Standards, May 1987.

[5] *Information processing systems - Open Systems Interconnection - Management Information Protocol Specification - Part 2: Common Management Information Protocol.* International Draft Proposal 9596-2, International Organization for Standards, Draft of 13 November 1987.