Agent Extensibility via the EMANATE Framework

Jeffrey D. Case
President, SNMP Research, Inc.

March 29, 1994

NOTES

EMANATE Goals

EMANATE: Enhanced MANagement Agent Through Extensions

- Designed to address the problem of multiple agents on a single platform via a modularly constructed extensible agent to meet the needs of:
  - multiple development teams within company / avoiding version skew and problems of "The BIG build"
  - customer extensions
  - third party software developers

NOTES

Goals: Design center

"Plug and play" MIB modules

- Traditional network management
- System management
- Application management, i.e., management of transient desktop applications
- Proxy management
- Manager-to-Manager communications
- Hot swap serendipity

NOTES

Objectives of the EMANATE Project

- conformance to the relevant Internet-standard Management Framework specifications, including:
  - SNMPv1
  - SNMPv2
  - Bi-lingual approach implemented
- ease-of-use for subagent developer
  - complexity in master agent / simple subagents
  - cookbook approach with heavy use of MIB compiler tools and automatic code generation
- modularity/portability to a wide range of target environments
  - minimal prerequisites
  - performance optimizations

NOTES

Comparison With Other Approaches

SMUX (RFC 1227) was the first generation, DPI (RFC 1228) was the second generation, and EMANATE is the third generation

- Similar to SMUX in goals and purpose but very different mechanisms
- Much lower overhead
  - smaller code size
  - more efficient / faster / lower CPU processing cost
  - easier for sub-agent writers
- SNMP compliant
- Similar to DPI in simplified API approach – the best part of DPI

NOTES

Approaches

In order to have a compliant extensible agent, it needs to be two things:

- compliant
- extensible

Note that it is much more feasible to make a compliant extensible agent from a compliant non-extensible agent than from a non-compliant extensible agent, e.g., SMUX or DPI

NOTES

Enrollment Requirements

Differentiation by:

- objects (this is easy – everybody does this)
- object instances
- priorities
- geographical and temporal domains (contexts)
- combinations and permutations of the above

AND do the above in the same or a different subagent

This is not-so-easy to do correctly

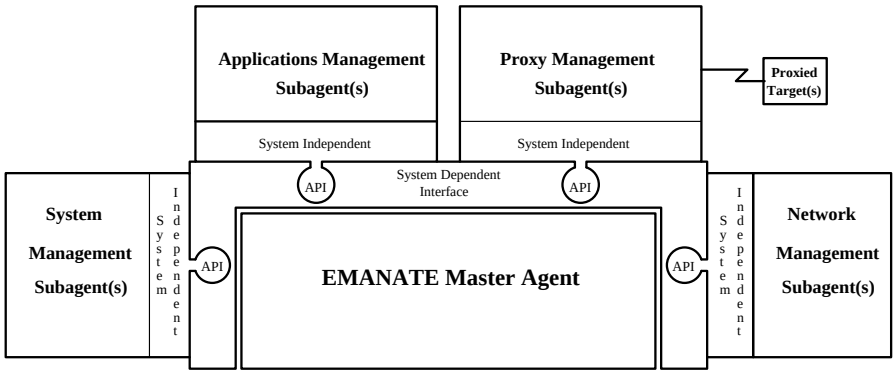
NOTES

EMANATE Architecture

- Master Agent / Subagent Paradigm
- Master Agent: agent protocol engine, security, privacy, security configuration, possibly more [but typically not]
- Subagents: simple and easy to implement and test
- Asynchronous message passing interface
- Master agent and subagents may be started in any order

NOTES

EMANATE Architecture



NOTES

Master Agent

MIB independent
Protocol (SNMPv1 / SNMPv2) dependent

- Core agent engine
- Supports SNMPv1 and/or SNMPv2 via compilation options
- Authentication, authorization, access control, and privacy
- EMANATE agent is multi-threaded / asynchronous
- Each VariableBinding (VarBind) results in zero, one, or many calls to method routines via the dispatch table (typically one or two)
- Contains all of the "hard parts"

System independent and system dependent portions

NOTES

Master Agent (Continued)

- System dependent portions:
  - network packet initialization, receive, send routines, e.g., UDP interface for traffic from / to manager station(s)
  - system dependent portion of the master agent end of the master agent/subagent interface
  - support for multi-threading
  - non-volatile storage access routines
- System independent portions include everything else:
  - get-next and get-bulk processing
  - multi-phase set processing
  - most of trap handling
  - enrollment/de-enrollment/lookup
  - subagent management

NOTES

Subagents

MIB dependent
Protocol (SNMPv1 / SNMPv2) independent

- zero, one, or many subagents per Master Agent
- built using Subagent Development Kit (SADK)
- modular: system independent and system dependent portions
- subagents primarily consist of Method Routines conforming to the System Independent Application Programming Interface (API)
- Subagent method routines have the same API as method routines in Master Agent and Monolithic Agent, thereby easing code and tool re-use

NOTES

Dispatch Table

Array of ObjectInfo structures:

```
typedef struct _ObjectInfo {
    OID             oid;
#ifndef LIGHT
    char           *oidname;
#endif                              /* LIGHT */
    unsigned char   oidtype, oidstatus, nominator, priority;
    VarBind        *(*get) ();
    int             (*test) ();
}               ObjectInfo;
```


NOTES

Dispatch Table (Continued)

- organized in lexicographic order for efficient accesses
- generated automatically by the postmosy phase of the MIB compiler
- constructed in the Master Agent from Dispatch Table components provided by Subagents at enrollment time
- dispatch table is pruned as subagents exit or die

NOTES

Method Routines

- read routine: used for get / get-next / get-bulk
  with or without caching supported
- write routines: multi-phase sets (5 passes, 4 methods)
  - test method
  - set method
  - cleanup method
  - undo method (possibly NULL)

NOTES

Method Routines (Continued)

- each routine has a system independent portion, and a system dependent portion
- the purpose of the system dependent portion to access the instrumentation and translate the system-dependent data structures into system-independent data formats specified by the MIB
- interface between the two is a data structure generated by the MIB compiler
- most (sometimes all) of the system independent method routines are generated automatically by the postmosy phase of the MIB compiler
- stubs are generated for the system dependent method routines

NOTES

Stub: Before

```
snmpFddiMACEntry_t *
k_snmpFddiMACEntry_get(int serialNum, ContextInfo *contextInfo, int nominator,
int searchType, long snmpFddiMACSMTIndex, long snmpFddiMACIndex)
{
   static snmpFddiSMTEntry_t snmpFddiSMTEntryData;

#ifdef NOT_YET
   snmpFddiMACEntryData.snmpFddiMACSMTIndex = ;
   snmpFddiMACEntryData.snmpFddiMACIndex = ;
   ...
   snmpFddiMACEntryData.snmpFddiMACFrameCts = ;
   snmpFddiMACEntryData.snmpFddiMACErrorCts = ;
   ...
   snmpFddiMACEntryData.snmpFddiMACFrameCondition = ;
   snmpFddiMACEntryData.snmpFddiMACChipSet = ;
   snmpFddiMACEntryData.snmpFddiMACAction = ;

   return(&snmpFddiMACEntryData);
#else /* NOT_YET */
   return(NULL);
#endif /* NOT_YET */
}
```


NOTES

Stub: After

```
    /* code to retrieve fa_mib not shown */
    ...
    snmpFddiMACEntry.snmpFddiMACSMTIndex = smtindex0 + 1; /* 1 based */
    snmpFddiMACEntry.snmpFddiMACIndex = macindex0 + 1;    /* 1 based */
    ...
    snmpFddiMACEntry.snmpFddiMACFrameCts = fa_mib->MACCountersGrp[0].Frame_Ct;
    snmpFddiMACEntry.snmpFddiMACErrorCts = fa_mib->MACErrorCtrsGrp[0].Error_Ct;
    ...
    snmpFddiMACEntry.snmpFddiMACFrameCondition =
        (fa_mib->MACStatusGrp[0].FrameErrorCondition) ? 1 : 2;
    snmpFddiMACEntry.snmpFddiMACChipSet = null_object;
    snmpFddiMACEntry.snmpFddiMACAction = D_snmpFddiPORTAction_other;
    ...
}
```


NOTES

Adding Agent Support for New MIB Objects

A 5 step process:

Step 1: MIB Definition: Select the MIB(s) to be supported by the Subagent

- Express in concise MIB format (RFC 1212) or SNMPv2-SMI format.
- Check for syntactic, semantic, and look-and-feel correctness.
- MIB compiler assists with some of the correctness checks.
- Use standard MIBs whenever possible.
- The most difficult and time consuming task.
- Time required proportional to the number of people helping.

Use MIB compiler tools to produce (for this example) demo.my

NOTES

Adding Agent Support for New MIB Objects (Continued)

Step 2: Build Subagent skeleton:

- edit Makefile to "name' the subagent, e.g., **MIB = demo**
- type "make" or "nmake" as appropriate

This produces a working (but still useless) subagent with no instances of any variable. Use this base for incremental program improvement.

NOTES

Adding Agent Support for New MIB Objects (Continued)

The following are produced:

- demosupp.h – supported MIB variables
- demotype.h – structure typedefs and function prototypes
- demopart.h – parts for the dispatch table data blocks ("under" the pointers)
- demooid.c – dispatch table fragment object blocks
- demodefs.h – manifest constants, e.g., enumerations
- v_demo.c – the system independent method routines
- k_demo.c – the system dependent method routines
- demoagt.c – the subagent
- demoagt – the linked, runable executable

Can use undefs.h to eliminate anything unsupported/unsupportable

NOTES

Adding Agent Support for New MIB Objects (Continued)

Step 3: Create the Instrumentation / Integrate with the Instrumentation

- For example, add a gauge to indicate the number of valid entries in a database cache
- May be greatly simplified (or bypassed entirely) if augmenting a proprietary management protocol or instrumentation already exists
- integrate with I/O dispatch in instrumentation, if any k_initialize() and k_terminate() can assist with this

NOTES

Adding Agent Support for New MIB Objects (Continued)

Step 4: Complete the Access Method Routines.

- edit k_demo.c as necessary to retrieve and transform data structures into the machine-independent format specified by the MIB
- get, get-next, and get-bulk then will be operational
- edit v_demo.c to include range and size checks in test functions and complete k_demo for read-write variables and notifications (traps and v2 informs)

NOTES

Adding Agent Support for New MIB Objects (Continued)

Step 5: Test

- Often the second most difficult step
- Testing the instrumentation is usually harder than testing the agent

NOTES

Status

- Implementation
  - Platforms
  - MIBs
  - Adapters
- Standardization, a.k.a., Market acceptance

NOTES

Implementation Status: Platforms

EMANATE's portability has been well proven:

- Several dialects of Unix: SunOS (Sun), Ultrix (DEC), HP/UX (HP), DG/UX (Data General), AIX (IBM), Dynix (Sequent)
- Several dialects of System V: Solaris (Sun), ODT (SCO), ESIX
- DOS, DOS/Windows, and DOS/Windows/Windows sockets (WinSock)
- Several embedded systems: VxWorks, pSOS, proprietary
- OS/2 (not latest version)

NOTES

Implementation Status: MIBs

EMANATE's capability has been well proven:

- System Management Subagents:
  MIB 2, Host Resources MIB, Proprietary
- Network Management Subagents:
  FDDI, Ethernet Repeater, X.25, etc MIBs
- Application Management Subagents:
  Database, emacs, Client-Server Apps
- Proxy Management Subagents:
  UPS MIB, legacy systems via vendor MIBs
- Mid-level Management ("Thinker") Subagents:
  RMON, M2M, Script MIBs

NOTES

Implementation Status: Adapaters

EMANATE's adaptability has been well proven:

- SMUX adapter for IBM
- DMTF adapter
- Native agent adapter

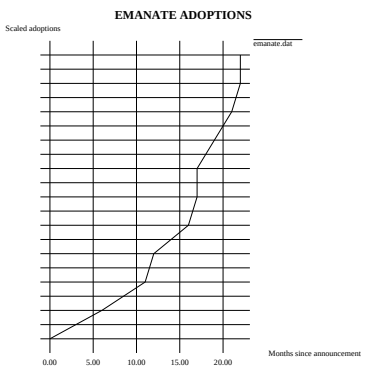
NOTES

Standardization Status

- need for this technology was recognized years ago
- when frustrated when the IETF declined to consider standardization in this problem-space, we took care of it ourselves
- we engaged another standardization body: the MARKET
- a *de facto* standard is, as always, being selected by the market
- this has been a slow and time-consuming effort for us but we have turned the corner since the Houston IETF

NOTES

Market Acceptance



NOTES