Contents Index

6

Fine-tuning Your Systems

ENlighten/DSM provides you with two powerful tools that allow you to fine-tune and manage your network of systems. The Programmable Events Processor (PEP) allows you to centrally manage events so that it reports only on issues you predetermine.

The Enterprise Management Database (EMD) is an application manager in charge of all communication between the database and other ENlighten/DSM components, an Informix ODBC driver, and an Informix database.

This chapter describes how to use these two tools.

Dispatching and Managing New Events

ENlighten/DSM has a Programmable Events Processor (PEP) for dispatching and managing new events. You can use PEP as a central point of event control, so it only reports on issues you want to see. You can also use PEP to communicate with your own external applications, such as Remedy.

PEP is implemented in ENlighten/DSM as a daemon. There must be at a minimum of one instance of PEP per network, but no more than one instance per host machine. Most installations of ENlighten/DSM will use a single PEP daemon, or a primary and secondary PEP daemon.

Applications can inform PEP of noteworthy events by using an Remote Procedure Call (RPC)-based Application Programmers Interface (API). PEP will then perform whatever action(s) you've previously defined for that particular event. PEP can also log these events in its common logging facility.

The diagram in Figure 6-1 shows how PEP interacts with the other main components of the ENlighten/DSM package.

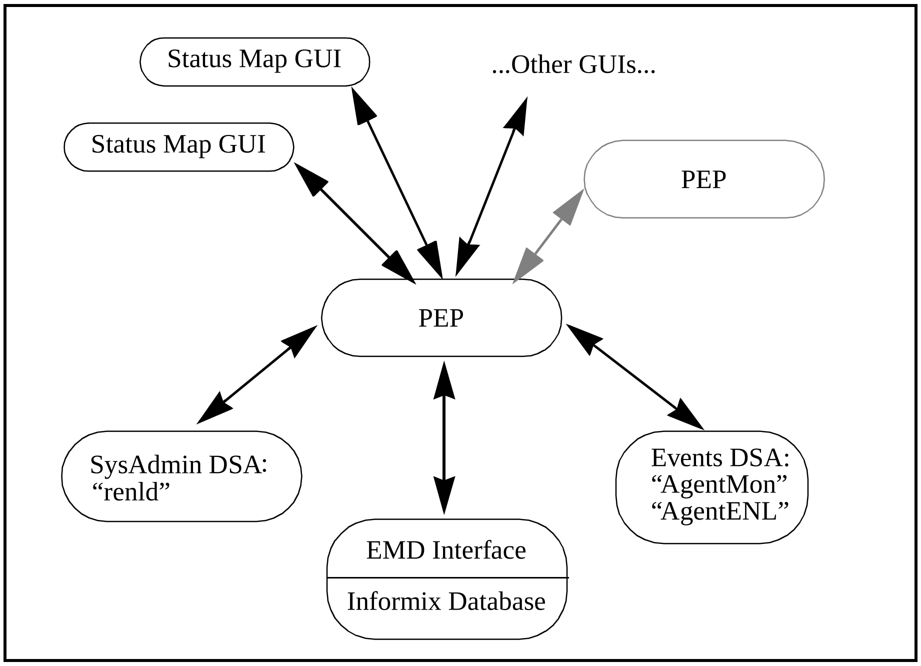

Figure 6-1 PEP interaction with ENlighten/DSM components

Functionality

PEP has a central role determining how the parts of ENlighten/DSM work. It can do the following:

- Accept event notifications via an RPC-based API.
- Write events to the common log via the Enterprise Management Database (EMD).
- Determine how to dispatch the event (using a rule-based policy engine).
- Dispatch events.
- Manage multiple Status Map GUIs reflecting the event activity.

- Manage the actions between GUIs, for example, acknowledging an event on one GUI will cause it to be acknowledged on all other GUIs.
- Forward events to another PEP (see Figure 6-1), allowing scalability for large networks.
- Help you query for an event's relationship to pools.

Policy Engine

PEP uses a rule-based dispatch table as its policy engine. Events are dispatched based on an ordered matching scheme. You can also add rules to define when this matching stops.

Each incoming event is matched against all defined rules (unless a stopmatch command is encountered). Events can be matched based on the following fields:

- Actual event ID (test name)
- Host where the event was generated
- Severity level of the event
- Application class of the event
- Application type
- Time the event occurred
- The Description of the event

Actions

Based on the policy you define (or the defaults), PEP can take one or more of the following actions:

- Send e-mail
- Send notification to the Status Map
- Start a process you defined and run it on any managed host
- Log the event
- Forward the event to another instance of PEP (via RPC)
- Query for the relationship to a pool

Status Map GUI Dispatching

PEP maintains connections with all running GUIs to allow the inter-operability between them. This includes:

- Maintaining a list of current on-line GUIs
- Dispatching Events to all GUIs
- Marshalling Events' acknowledgments from the acknowledging GUI to all other connected GUIs

Policy Definition

The policy definition is a rules-based system contained in an ASCII file, so rules can be defined and changed dynamically as the system grows. Rules are scanned in the order they are defined.

Rules-Based Language

Using this rules-based method, new events can be generated and complex relationships can be created between events. The policy engine supports the following elements:

- Complex, nested if/then/else syntax
- Variables (local, global, static, and dynamic)
- Timers with one-second granularity
- Pattern matching on string fields
- Time of day, day of week, date matching
- Modifying the elements of the current event
- Querying for pool relationships

You can use timers to set up rules based on an occurrence of an event multiple times within a given time period. Portions of an event can be modified, as shown in the following example. This effectively creates a new event from an existing one, which can then be further dispatched.

Common Logging

All data logs go directly into the EMD. All event logs go through PEP for additional dispatching.

```
# rules for various events

rule    log all events
    event -> log
;
rule    all clear # always send all clear events to gui
    if event.severity = allclear
    then
        event -> gui
    endif
;
rule    severe events # tell map and send mail or page
    if event.severity >= severe
    then
        event -> gui    # notify the GUI
        if timeofday >= 8am and timeofday <= 5pm
        then
            event -> mail “bob@dispatch”
                     # send email to the dispatcher
        else
            event -> exec “localhost” “/etc/page” “456-1234”
        endif
    endif
;
rule    rpc client bad calls # create new event
    variable bad_calls initializer 0 # a local variable
    if event.host = “taz” and event.name = “rpc.badcalls”
    then
        increment bad_calls
        if bad_calls > 10
        then
            event.severity = severe
            event.hostname = “PEP”
            event.explanation = “10+ rpcc client bad calls on taz!”
            event -> gui
            event -> log  # original event was already logged
            set bad_calls 0
            stopmatch  # don’t match against any more rules
       endif
    endif
;
```

Manually Killing/Invoking PEP

Sometimes you may want to manually kill or invoke PEP, for example, when you're bringing down the system for maintenance or upgrading to a new release of ENlighten/DSM.

☞ You *must* shut down EMD and you should shut down PEP before backing up any database tables and then manually restart them afterward.

The rest of this section details how to do this for PEP.

Terminating PEP

You can bring down the `pep` daemon with the following CLI command:

```
kill -2 <pid#>
```

This sends an interrupt to the daemon and allows it to shut down gracefully.

Starting PEP

You can invoke the `pep` daemon with the following CLI command:

```
./pep &
```

This starts the daemon in the background. If you have changed the pathname for the `rules.txt` file (from `$ENLIGHTEN/contrib/rules.txt`), you need to use the following command instead:

```
./pep -f yourpathname/contrib/rules.txt &
```

where *`yourpathname`* shows the full pathname to the `rules.txt` file.

Program Layout

PEP programs must adhere to a certain layout. The general layout is:

```
# Comments start with "#" and
# Terminate at the end of the line
#
Global variable declarations
Timer declarations

Rule declaration # Comments can start any where
   Local Variable Declarations
   Program
;

Rule declaration
   Local Variable Declarations
   Program
;
```

More specifically:

- The program allows comments to begin anywhere on the line and extend all the way to the end of the line.
- Any global data must be declared before any rules are defined.
- Timers are implicitly global. They must come after any global variable declarations.
- Rules are terminated with a semicolon (;). You can use as many rules as necessary.

Program Execution

The engine program is executed every time PEP receives an event. External processes, such as SysAdmin agents or Events agents, can send an event using timers. PEP can also do this (see "Timers" on page 6-11 for more details).

There is a special event called a start-up event that gets generated internally at start-up. You should use this event to initialize timers.

Data Types

The PEP language supports several different data types:

- integer numbers
- floating point numbers
- strings
- date/time
- timers
- events
- identifiers
- constants

Integer Numbers

The following are examples of valid integer declarations:

```
-1  32  42235523  09
```

The following are examples of invalid integer declarations:

```
+24  +2442533
```

Floating Point Numbers

The following are examples of valid floating point number declarations:

```
4.2   0.332   -.442  -2.44   -0.243   89.0
```

The following are examples of invalid floating point number declarations:

```
3.2e02  +5.2
```

Strings

Strings are anything in quotes (“”) except the quote character itself. All string data types in PEP support regular expression matching, including wildcard matching. The following are examples of valid string declarations:

```
“Hello There”
“He said *”
“123”
“[a-z].*”
“I can punctuate...,,,”
```

The following are examples of invalid string declarations:

```
“what “?”
“ No end in site
was”sup
```

Date/Time

PEP supports partial time and date declarations. You can use the keyword timeofday to read the system’s local time. Add the designators PM/AM to specify absolute time designation. This is what the engine looks for to distinguish between relative and absolute time references.

Dates are declared in U.S. notations and assume the local time of the host system. The notation is as follows:

```
month day year OR
month year
```

It does not support the European or Asian date notation of:

```
day month year OR
year month day
```

A few different styles of notations and separators are supported. The following are examples of valid date/time declarations:

```
12/95   12-95   12/3/1990   12-3-90
timeofday
Dec 4, 1992
Wed 9:00 PM
3 am
12:24:31 AM
Tuesday
```

The following are examples of invalid date/time declarations:

```
12:30
1995/12/3
1:1
15/7/96
```

Timers

Timers are special data types that are an extension of the date/time data type. Timers support relative and absolute time specifications. Relative time is either *hr*:*min*:*sec* or *min*:*sec*. The following are examples of valid timer declarations:

```
0:30   1:00:00   1:00
```

Remember that timers support all date/time declarations as well. Timers can also be manipulated programmatically. They have two fields:

Field name	**Data type**	**Description**
state	string	Current state of the timer (Set, Cleared, Expired). See "Constants" on page 6-13.
time	timer	Absolute or relative time of timer.

Events

Anything that is sent to PEP is considered an event. PEP itself can generate events internally, one on start-up and one for every timer expiration. Events have components you can inspect or modify. These fields are designated with the dot `'.'` operand. The fields are:

Field name	Data type	Description
severity	constant	Severity level of the event.
name	string	Testname that generated the event.
hostname	string	Name of the host that caused the event.
description	string	Descriptive text about the event.
time	date/time	Time when the event was generated in system local time.
application	constant	Application type that sent the event.
appclass	constant	Application class that sent the event.

Identifiers

You can use identifiers to name variables, rules, and timers. They must begin with a letter and can contain letters, numbers, or underscores ('_'). The following are examples of valid identifiers:

```
a1234  bbc  ABC  A_1234
```

The following are examples of invalid identifiers:

```
1A  A-Z  bor%d
```

As a general rule, you should use lowercase letters for identifiers, since constants start with uppercase letters. This will help avoid confusion between identifiers and constants.

Constants

You can use constants to compare event subfields and timer states to preset states. Constants are actually strings and integers that are hardwired to a certain state. All constants begin with an uppercase letter. The following is a list of constants and their appropriate uses:

Events have one of these five severity types:

```
Okay
Info
Severe
Warning
Error
```

Events have one of these seven application types:

```
EventAckOne
EventAckAll
EventDeletePool
EventMoveIcon
EventChangeMap
EventInit
EventTimer
```

Events have one of these three application classes:

```
EventClass
AdminClass
CLIClass
```

Timers have one of these three states:

```
Set
Expired
Cleared
```

Program Syntax

The PEP language consists of:

- variables
- timer declarations
- rules declarations
- expressions
- implicit conversions

Variables

Variables can be either dynamic or static. All variables must be explicitly assigned and are implicitly typed:

```
variable static x = 2
```

This variable declares a static variable x of an integer type and assigns the value of 2 to it. The following are more examples of valid variable definitions:

```
variable static y = 4.2 + 44.2
variable cc = “Hello World”
variable backup_time = 10:30 pm
variable now = timeofday
```

The variable’s domain is implicit where it is declared. Global variables are always defined before any rule declarations and are accessible in any rule. Local variables are always declared after the rule declaration and only accessible to the rule that declared it. Variables that are named the same as global ones will hide the globals and continue until they go out of the local scope.

Timer Declarations

You can use timers to set the time-in and time-out of events in PEP. Time-outs generate events themselves. Timer states are implicitly set to “CLEAR” and must be initialized explicitly using the start-up event:

```
timer backup = 4:30 am
timer april_fools = Apr 1
timer one_minute = 0:60
timer one_a_day = 24:00:00
```

Rules Declarations

Rules are declared with the key word rule and followed by a name using a valid identifier. Rules must have a body:

```
rule myrule
   variable static count = 0
   if ( event.severity > 0 )
   then
      increment count
   endif
;
```

The rule `myrule` uses a local static variable to count the number of events that have a positive severity.

Expressions

Expressions can be assignments, decisions, and send event statements.

Assignments

Assignments are performed by using the equal sign (**=**). The right-hand side can be a complex expression. The left-hand side must be a variable or part of the event structure.

```
x = y + 432
event.hostname = “New hostname”
```

The following arithmetic operations are supported: **+**, **-**, **/**, *****, **%**, increment, decrement.

For example:

```
z = 42.3 - 24.0 * 2.0
x = z / 42.1
b = 5 + 4
increment b
decrement c
```

Decisions

You can make decisions by using an "if/then/else/endif" expression in conjunction with logical expressions (==, !=, <, >, <=, >=, and, or). An if expression must have a then and a terminating endif statements; the else branch is optional.

```
variable count = 0
variable myval = 1

if ( event.hostname == “a*” )
then
   increment count
   if ( count >= 2 and my_val < 3 )
   then
      increment myval
   else
      decrement myval
   endif
endif
```

Use the stopmatch command to stop a rule's execution. When PEP encounters a stopmatch command, it will "break out" of the program and discontinue execution until the next event occurs.

Send Events

Events can be sent to five separate targets (log, gui, mail, exec, pep) via the “**->**” notation.

```
event -> log # event is forwarded to logger
if ( event.severity > 0 )
then
   event -> gui # event is forwarded to all guis
else
   event -> mail “user@company.com”
endif
event -> exec “aShellCommand arg1”

# forward the event to the secondary PEP at host
# masterhost
event -> pep at “masterhost”
```

Whole events can be sent to different targets this way. The log and gui targets don’t require any additional arguments. The mail target must be a user specified in quotes (for example, “jk“). The pep target must be followed by the key phrase ‘at hostname’.

The exec target must contain the command to execute and any arguments in quotes (for example, “aShellCommand arg1“). The values of the event are passed to the shell as environment variables. The Events fields are listed in the table below.

Event field	**Corresponding environment variable**
name	$ENL_TESTNAME
hostname	$ENL_HOSTNAME
severity	$ENL_SEVRITY
description	$ENL_DESCRIPTION
time	$ENL_EVENTTIME

Event fields and timer fields can be modified individually, as shown in the examples below:

```
timer a = 0:00:30 # 30 second timer

rule startup_rule
   if ( event.name == “EventInit” )
   then
      a.state = “Set”   #set the timer on startup
   endif
;
```

Query Events

PEP supports simple queries. Use the following syntax:

```
host in pool abc_pool
```

This command searches the pool *abc_pool* recursively to see if the *host* is in that pool or its subpools.

This is useful when you set up your own pools and subpools. You can write a rule to check if an event came from a certain pool of hosts. For example, the following code fragment will check if an event came from the pool “important_pool”:

```
if ( event.hostname in pool “important_pool” )
then
   event -> mail “user”
endif
```

Implicit Conversions

PEP supports manipulating different data types by using implicit data type conversion. Unlike C, which converts to the “highest” data type, PEP converts based on left-to-right evaluations of expressions.

Simple Comparisons

A simple example is:

```
variable a = 2
variable c = 4.2
variable i = 0



if ( a + c > 6 )
then
   increment i
endif

if ( c + a > 6 )
then
   increment i
endif
```

In the first if expression, `a + c` is evaluated as an integer added to a float. Since the integer is the first expression, *c* is also converted to an integer. The temporary result is an integer with a value of 6, thus causing the if expression to evaluate to be false.

In the second if expression, `c + a` is evaluated as a float added to an integer. Since the float is the first expression, *a* is converted to a float. The temporary result is a float with a value of 6.2. This temporary float also causes the constant 6 in the if expression to promote to a float of 6.0, thus causing the if expression to evaluate to true.

Date/Time Comparisons

Absolute date/time comparisons compare only the common elements defined. For example, variables *a*, *b*, *c*, and *d* are set as follows:

```
variable a = 12/19/95
variable b = 2 pm

variable c = Tuesday
variable d = timeofday
```

Assume the current time is “Tuesday Dec 19, 1995 2:25pm”. Comparing variable `a`, `b`, or `c` to variable `d` would evaluate to true because each comparison only compares the common date/time components. However, comparing `a` to `b`, `b` to `c`, or `a` to `c` would result in a false comparison since none share common components. If we added an additional variable `e`,

```
variable e = 2:25 pm
```

comparing `e` to `b` would result in a true expression since the only common components are the hour and am/pm designator.

Unsupported Conversions

The “string to time” conversion is not supported at this time.

Program Start-up

When the program is started, a “startup” event is passed through the system. The event’s name is set to the constant `EventInit`. This event should be used to explicitly initialize data such as timer states.

PEP Engine Program Example

The following is an example of a PEP engine program to test the event e-mail process:

```
# test program

variable appServer = “Hostname.*”
variable collisions = “Testname.*”

rule alwayslog
   event -> log
;

rule testit
   if ( event.severity == Okay )
   then
      event -> gui
   endif
;
rule severe_check
   if ( event.severity == Severe )
   then
      event -> gui
      if ( timeofday >= 10:00 am and timeofday <= 10:23 am )
      then
         event -> mail “jk”
      else
         event -> exec “ls -lt > sh.$$”
      endif
   endif

rule ss
   variable static collision_count = 0

   if ( event.hostname == appServer and event.name == collisions )
   then
      increment collision_count
      if ( collision_count > 10 )
      then
         event.severity = severe
         event.hostname = “PEP”
         event.description = “More then 10 collisions”
         event -> gui
         event -> mail “jk”
         collision_count = 0
         stopmatch
      endif
   endif
;
```

Enterprise Management Database

The Enterprise Management Database (EMD) is in charge of all communication between the database and other ENlighten/DSM components. It consists of three components:

- Informix SE Database Engine with an ANSI-compliant database.
- ODBC drivers (currently supporting only Informix)
- A daemon emdd that handles all requests from other ENlighten/DSM components.

As an RPC-based server, the emdd communicates with the following ENlighten/DSM components: Events agent, SysAdmin agent, Status Map, PEP, and the user interface.

The communication layer between emdd and the database is ODBC compliant. Our ODBC drivers are supplied by Visigenic Software Inc. The drivers are Core and Level 1 API conformant and Minimum and Core SQL conformant.

The emdd daemon also checks daily to see if any data in the database needs to be expired. See Chapter 3, “Configure,” in the *ENlighten/DSM Reference Manual* for more details.

Relational Database

ENlighten/DSM includes a relational database as part of EMD. The following data is stored in the database:

- pool configurations
- session preferences
- host overrides
- user authorizations
- add user templates
- archive device configurations
- Events log data
- Events alarm data
- software and hardware inventory lists generated by Events

- acknowledgment of events from the Status Map
- backup catalogs
- scheduling of backups
- host notes

The following data is not stored in a database:

- Events host-specific `testtab` files
- Events `AgentENL.config` file data
- snapshots of disks/file systems

Directory Structure

The EMD directory structure is:

- *`install-path/bin`*
- *`install-path/dbtables`*
- *`install-path/odbc`*
- *`install-path/informix`*
- *`install-path/msg`*
- *`install-path/log`*

`where:`

`bin`	contains the emdd daemon and scripts such as start_enl daemons
`dbtables`	contains the database tables, the database's transaction log, and a script to create the database tables
`odbc`	contains the ODBC drivers and other related components
`informix`	contains the Informix SE products
`msg`	contains any message catalog files
`log`	contains the emdd’s logfile emdd.log

Database Access

The database and its tables are owned by the user dbenl. This is a new user that is created at installation time. User dbenl's home directory will contain an odbc initialization file `.odbc.ini`, and its `.cshrc` or `.profile` file will contain the environment variables necessary to run Informix and the ODBC driver.

By default, only user dbenl has access to the database. dbenl must start the daemon emdd or the daemon will not be able to access the database. The database administrator at your installation site may change the database privileges.

Warning! Do not modify or rename the `.odbc.ini` initialization file. Doing so will make all subsequent EMD usage and connections fail.

Manually Killing/Invoking EMD

Sometimes you may want to manually kill or invoke EMD; for example, when you're bringing down the system for maintenance or upgrading to a new release of ENlighten/DSM.

☞ You *must* terminate EMD before backing up any database tables and then manually re-invoke them afterward.

The rest of this section tells how to do this for EMD.

Terminating EMD

You can bring down the `emdd` daemon with the following command:

```
stop_enl_daemons
```

This sends an interrupt to all ENlighten/DSM daemons and allows the program to shut down gracefully. The daemon waits for any child processes to exit before shutting down. This may take up to 10 seconds.

Starting EMD

You can invoke the emdd daemon with the following CLI command:

```
start_enl_daemons
```

This starts the daemon in the background and sets the log level to 0. The log file is generated, but only error messages are logged (if any occur).