HP OpenView Operations for UNIX White Paper

Software version: A.08.12

Getting started with XML/Perl Programming for the OVO Service Engine

Written by Vesna Soraic

Abstract 2
Background 2
Preparation 3
Query all services with status 4
Register and listen for status changes 7
Create additional labels on services 9
Remote access 10
Summary 11
Appendix 12
For more information 15
Call to action 15

Abstract

HP OpenView Service Navigator is an add-on component of the HP OpenView Operations Java-based operator GUI. It enables you to manage your IT environment while focusing on the IT services you provide.
The HP OpenView Service Navigator service engine (opcsvcm) is the backend for the Service Navigator. It maintains the server model and calculates the status of services. To provide programmatic access to the service engine, HP OpenView Operations contains an XML data interface that allows you to write or get service configuration directly into or from the service engine.
This paper discusses how you can use XML and Perl to programmatically access the service engine so that you can enhance or automate Service Navigator task. We will use the following examples:

- List services and their status
- Register for statues changes
- Add labels to services

Background

opcsvcterm(1M) is the interface to HP OpenView Service Engine which inputs XML into `stdin` and outputs XML to `stdout`. The XML Data Interface servers uses cases such as:

- Allows you to write the service configuration directly into the service engine. The configuration syntax follows the XML rules defined in the document type definition (DTD) `operations.dtd`.
- Allows you to get the current service configuration and service status directly from the service engine. The output syntax follows the XML rules defined in the DTD `results.dtd`.

The format of the operations and results files is based on the World Wide Web Consortium Extended Markup Language (XML). The DTDs for the Service Navigator XML syntax are printed in this section and are also available on the OVO management server at the following location:

```
/etc/opt/OV/share/conf/OpC/mgmt_sv/dtds/services.dtd
/etc/opt/OV/share/conf/OpC/mgmt_sv/dtds/operations.dtd
/etc/opt/OV/share/conf/OpC/mgmt_sv/dtds/loggings.dtd
/etc/opt/OV/share/conf/OpC/mgmt_sv/dtds/results.dtd
```

All DTDs are also available in the XML Schema Definition (XSD) format in the same directory. This alternative format is based on XML and therefore easier to read with XML editors.

©Copyright 2005

Preparation

For running the examples described in this paper you need an OVOU server where the service engine (opcsvcm) is installed and running. No additional software is required.

In case you do not have any services defined yet, you may add some example services. Add the email example using following command:

```
# opcservice -add /opt/OV/OpC/examples/services/email.xml
```

Since we want to verify our XML and Perl with the Service Navigator, we have to assign the email service to an user. Enter the command:

```
# opcservice -assign opc_adm email
```

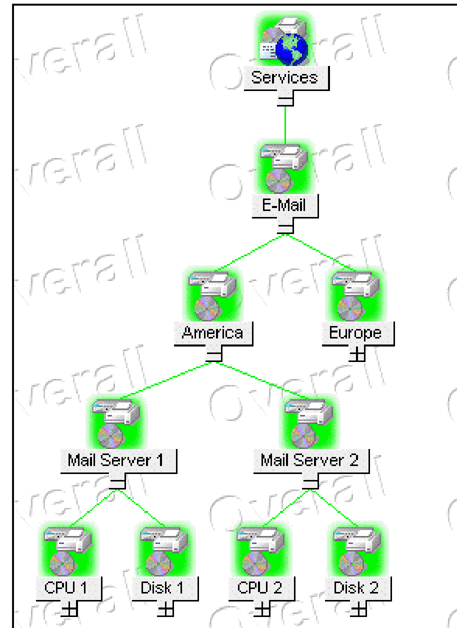
To see what kind of top level services are in the service engine, enter the command:

```
# opcservice -list
```

This command returns:

```
...
Service: email
...
```

When you open the Service Navigator, the email service tree like it is displayed as illustrated here:



Note, the following example makes use of `opcsvcterm(1m)`. If you have installed the Service Navigator value Pack, `opcsvcterm(1m)` will not work, and you must use the `cadmsnd(1M)` command instead.

©Copyright 2005

Query all services with status

With this 1st example we are going to access the service engine using XML and `opcsvcterm(1m)`. We will see how you can programmatically retrieve the status of services.

First, let us have a look at the XML tags we need to list services. Following XML sequence instructs the service engine to list all services:

```
<Operations>
   <List>
         <All/>
   </List>
</Operations>
```

Do this manually with `opcsvcterm(1m)`. In a shell, enter the command:

```
# opcsvcterm
```

`opcsvcterm(1m)` connects to the service engine and returns:

```
<?xml version='1.0' ?>
```

Now instruct the engine to list all services. Enter the following commands:

```
<Operations>
<List><All/></List>
```

Note, the service engine responses with a *<Results>* tag to the **<Operations>** instruction. The output of the list command takes the following form:

```
  <Services>
    <Service>
      <Name>node3_cpu</Name>
      <Status><Normal/></Status>
      ...
      ...
  </Services>
</Results>
```

We shall gracefully end the connection to the service engine by submitting the **</Operations>** end tag as follows:

```
</Operations>
</Results>
#
```

Now we will do the same listing of all services, but programmatically within a Perl script. We will first walk trough a couple of code extracts to see how we can talk to the service engine using Perl. We do not use any XML libraries or modules so that the examples are as simple as possible.

©Copyright 2005

Connect to the service engine using Perl.

The code fragment below uses the `open2` of the `IPC::Open2` module to create a connection to the service engine. Additionally, is parses the response by reading one line from `<$IN>` and verifies that the engine responded `with` *`<?xml version='1.0' ?>`*.

```
#!/opt/OV/bin/Perl/bin/perl

use IPC::Open2;

# Connect to service engine, open streams
#   OUT: write to the service engine
#   IN: read from the service engine

open2($IN, $OUT, "opcsvcterm");

$line = <$IN>;

if ($line =~ /xml/)
{
  print "connected\n";
}
else
{
  die "Cannot connect to service engine";
}
```

The program will stop if no connection to the service engine could be established.

Write operations into the service engine

Now we are ready to write commands into the service engine. As in the manual example above, we will write **`<List><All/></List>`**, but first we have to send the **`<Operations>`** tag.

```
# Send query to service engine
print $OUT  "<Operations>\n";
print $OUT  "<List><All/></List>\n";
```

Read responses from the service engine

The next code fragment describes how you could read results from the engine and parse the output to find certain XML tags. We are searching for the following XML text to get the service name and status from all service objects:

```
<Service>
  <Name>service_name</Name>
  <Status><service_status/></Status>
```


©Copyright 2005

Here we loop through the results until the service engine returns the `</Services>` end tag. The program concatenates incoming lines until it finds a `</Service>` end. Then it pattern matches to find the service name and status.

```
$service = "";

while ($line = <$IN>)
{
  chomp $line;
  last if ($line =~ /\/Services/) ;

  $service = $service . $line;

  if ($service =~ /<\/Service>/)
  {
    $service =~ \
/<Service>\s*<Name>(.*)<\/Name>\s*<Status><(\w*)\/><\/Status>/;

    print "Service:$1 - Status:$2\n";

    $service = "";
  }

}

print $OUT  "</Operations>\n";
```

Run the example program

The example we discussed above is listed in the appendix of this document. Copy the 1st example from the appendix and run it.

```
# ./example1.pl
connected
Service:email - Status:Normal
Service:america - Status:Normal
Service:email_node1 - Status:Normal
Service:node1_disk - Status:Normal
Service:node1_cpu - Status:Normal
Service:email_node2 - Status:Normal
Service:node2_disk - Status:Normal
Service:node2_cpu - Status:Normal
Service:europe - Status:Normal
Service:email_node3 - Status:Normal
Service:node3_disk - Status:Normal
Service:node3_cpu - Status:Normal
```


©Copyright 2005

Register and listen for status changes

In the 2nd example we are going to register and listen for status changes at the service engine using XML and `opcsvcterm(1m)`. You will see how you can get status change notifications programmatically.

First, let us have a look at the XML tags we need for this scenario. With following XML sequence we can register our program at the service engine so that the engine sends us status change notifications:

```
<Registration>
   <RegCondition>
         <ServiceRef>service_name</ServiceRef>
   </RegCondition>
</Registration>
```

Do this manually with `opcsvcterm(1m)`. In a shell, enter the following command:

```
# opcsvcterm
```

`opcsvcterm(1m)` connects to the service engine and returns:

```
<?xml version='1.0' ?>
```

Now register for changes of the **email_node1** and **email_node2** services. Enter the tags listed in **bold.** *Italic* tags are responses.

```
<Operations>
<Results>
<Registration>
<RegCondition><ServiceRef>email_node1</ServiceRef></RegCondition>
<RegCondition><ServiceRef>email_node2</ServiceRef></RegCondition>
</Registration>
  <OK/>
```

Now in another shell submit a major message for the `email_node1` service as follows:

```
# opcmsg a=a o=o msg_t=hello severity=major service_id=email_node1
```

You will see that your `opcsvcterm(1m)` session received following XML document:

```
<StatusChanges>
  <ElementStatusChange>
    <ServiceRef>email_node1</ServiceRef>
    <Status><Minor/></Status>
    <OldStatus><Normal/></OldStatus>
  </ElementStatusChange>
</StatusChanges>
```

Now let us wrap this example in some Perl code so that we can receive such status changes programmatically.
The connection to the service engine will be done the same way as in the 1st example and is not listed here. The code below shows how to register all services that are defined as command line options in `@ARGV`.

```
print $OUT  "<Operations>\n";
print $OUT  "<Registration>\n ";

 foreach my $service ( @ARGV )
  {
   print "Register for $service\n";
   print $OUT"<RegCondition><ServiceRef>$service</ServiceRef></RegCondition>";
  }
  print $OUT "</Registration>\n";
```

The next code extract describes how you read the status changes from the engine and parse the output to find the service name, and the new and old statuses. We are searching for the following XML text:

```
<ServiceRef>service_name</ServiceRef>
  <Status><current_status/></Status>
  <OldStatus><old_status/></OldStatus>
```

Here it loops through the results and concatenates incoming lines until it finds a `</OldStatus>` tag. Then it pattern matches to find the service name and the statuses.

```
      $service = "";

   while ($line = <$IN>)
   {
    chomp $line;

    # print "#line:$line\n";
    $one_change = $one_change . $line; #concat lines until OldStatus

    if ($one_change =~ /\/OldStatus/)
    {
       $srf="<ServiceRef>(.*)<\/ServiceRef>";
       $stat="<Status><(.*)\/><\/Status>";
       $ostat="<OldStatus><(.*)\/><\/OldStatus>";

       $one_change =~ /$srf\s*$stat\s*$ostat/;


       print "Severity of $1 service changed from $3 to $2\n";
       $one_change = "";
    }
   } # end while
```


©Copyright 2005

The program just prints the changes, but you can imagine that this example could be used for many other purposes like sending email notifications or submitting new messages via opcmsg that contain information about critical status changes of business services.

Run the example program

The example we discussed above is listed in the appendix of this document. Copy the 2nd example from the appendix and run it. First acknowledge all messages which you sent to the service `email_node1` previously.

```
# ./example2.pl email_node1 email_node2
connected
Register for email_node1
Register for email_node2
```

Now, in another shell submit a major message for the `email_node1` service:

```
# opcmsg a=a o=o msg_t=hello severity=major service_id=email_node1
```

The program `./example2.pl` writes the status change of the `email_node1` service:

```
Severity of email_node1 service changed from Normal to Major
```

Create additional labels on services

In the 3rd example we are going to create additional labels on services on the fly using XML and `opcsvcterm(1m)`. We will see how you can modify the content of the service engine dynamically. The example is rather simple, but can be extended to change more complex content like calculation rules to cover advance use cases.

First let us have a look at the XML tags we need for this scenario. With following XML sequence we can set a new label on a service:

```
<SetAttributes>
   <ServiceRef>service_name</ServiceRef>
   <Attribute>
         <Name>ov_label1</Name>
         <Value>label_text</Value>
   </Attribute>
</SetAttributes>
```

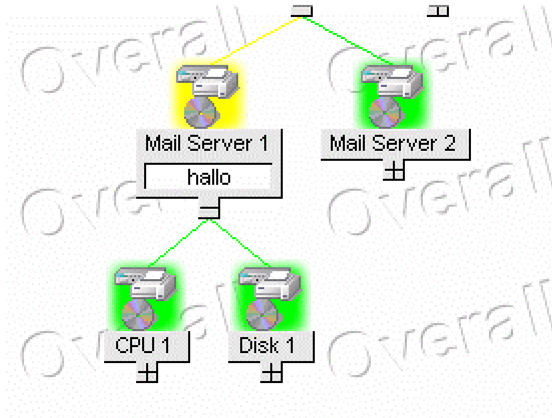
Run the example program

An example Perl program for changing labels is listed in the appendix of this document. Copy the 3rd example from the appendix and run it.

©Copyright 2005

```
# ./example3.pl email_node1 hallo
connected
OK. Set label hallo at service email_node1
#
```

A new label is added as it is shown in the picture below.



Remote access

It is also possible to access the service engine remotely. Your Perl program, which integrates with the service engine, can run on a different system than the OVO management server system. Before doing so, you need to enable the `opcsvcterm` port on your OVO management server system.

Edit `/etc/services` file and add:

```
opcsvcterm      7278/tcp        # Service engine remote access
```

Edit `/etc/inetd.conf` file and add:

```
opcsvcterm  stream tcp nowait root /opt/OV/bin/OpC/opcsvcterm opcsvcterm
```

Restart the inet deamon:

```
# /etc/inetd -c
```

You can test the connection to the remote `opcsvcterm` with a telnet command. You should see the following response:

```
# telnet ovoserver.demo.com 7278
Trying...
Connected to ovoserver.demo.com.
Escape character is '^]'.
<?xml version='1.0' ?>
```

Following code fragment shows how you can connect to a remote service engine using Perl:

```
use IPC::Open2;
use IO::Socket;
```

©Copyright 2005

```
$server = "ovoserver.demo.com",
$port = 7278;


if ($server eq "")
{
   print "Connect Local \n";
   open2($IN, $OUT, "opcsvcterm");
}
else
{
   print "Connect remote: \n";
   $remote = IO::Socket::INET->new(Proto    => "tcp",
                                   PeerAddr => $server,
                                   PeerPort => $port) or \
       die "cannot connect to port $port  at host $server";

   # in & out stream is socket;
   $IN= $OUT = $remote;
}
```

Summary

In this tutorial you can learn how to programmatically retrieve the service status and get status change notification and how to modify the content of the service engine dynamically. The appendix provides three simple programs that can be extended to cover more advanced use cases.

invent

Appendix

Example 1: List services and their status

```
#!/opt/OV/bin/Perl/bin/perl

################################################################################
#                    Warranty Information
# The information contained in this document is subject to change without notice.
# THE AUTHOR PROVIDES THIS MATERIAL "AS IS" AND MAKES NO WARRANTY OF
# ANY KIND, EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
# WARRANTIES OF MERCHANT ABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
# THE AUTHOR SHALL NOT BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR
# INCIDENTAL OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH THE FURNISHING,
# PERFORMANCE OR USE OF THIS MATERIAL WHETHER BASED ON WARRANTY, CONTRACT,
# OR OTHER LEGAL THEORY.
################################################################################

use IPC::Open2;

# Connect to service engine, open streams

open2($IN, $OUT, "opcsvcterm");

$line = <$IN>;

if ($line =~ /xml/)
{
  print "connected\n";
}
else
{
  die "Cannot connect to service engine"; }

   # Send query to service engine
   print $OUT  "<Operations>\n";
   print $OUT  "<List><All/></List>\n";

   $service = "";

   while ($line = <$IN>)
   {
     chomp $line;

     last if ($line =~ /\/Services/) ;

     $service = $service . $line;

     if ($service =~ /<\/Service>/)
     {
       $service =~
/<Service>\s*<Name>(.*)<\/Name>\s*<Status><(\w*)\/><\/Status>/;

       print "Service:$1 - Status:$2\n";

       $service = "";
```

©Copyright 2005

```
      }

    }

    print $OUT  "</Operations>\n";
```

Example 2: Register for statues changes

```
#!/opt/OV/bin/Perl/bin/perl
################################################################################
#                  Warranty Information
# The information contained in this document is subject to change without notice.
# THE AUTHOR PROVIDES THIS MATERIAL "AS IS" AND MAKES NO WARRANTY OF
# ANY KIND, EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
# WARRANTIES OF MERCHANT ABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
# THE AUTHOR SHALL NOT BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR
# INCIDENTAL OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH THE FURNISHING,
# PERFORMANCE OR USE OF THIS MATERIAL WHETHER BASED ON WARRANTY, CONTRACT,
# OR OTHER LEGAL THEORY.
################################################################################

use IPC::Open2;

# Connect to service engine, open streams
#   OUT: write registration to
#   IN: read status changes from

open2($IN, $OUT, "opcsvcterm");

$line = <$IN>;

if ($line =~ /xml/)
{
  print "connected\n";
}
else
{
  die "Cannot connect to service engine"; }


   print $OUT  "<Operations>\n";

   print $OUT  "<Registration>\n ";

   foreach my $service ( @ARGV )
   {
     print "Register for $service\n";
     print $OUT
"<RegCondition><ServiceRef>$service</ServiceRef></RegCondition>";
   }
   print $OUT "</Registration>\n";

   $service = "";
```

```
while ($line = <$IN>)
{
 chomp $line;

 # print "#line:$line\n";
 $one_change = $one_change . $line; #concat lines until OldStatus

 if ($one_change =~ /\/OldStatus/)
 {
    $srf="<ServiceRef>(.*)<\/ServiceRef>";
    $stat="<Status><(.*)\/><\/Status>";
    $ostat="<OldStatus><(.*)\/><\/OldStatus>";

    $one_change =~ /$srf\s*$stat\s*$ostat/;


    print "Severity of $1 service changed from $3 to $2\n";
    $one_change = "";
 }
} # end while


   print $OUT  "</Operations>\n";
```

Example 3: Add labels to services

```
#!/opt/OV/bin/Perl/bin/perl

################################################################################
#                   Warranty Information
# The information contained in this document is subject to change without notice.
# THE AUTHOR PROVIDES THIS MATERIAL "AS IS" AND MAKES NO WARRANTY OF
# ANY KIND, EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
# WARRANTIES OF MERCHANT ABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
# THE AUTHOR SHALL NOT BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR
# INCIDENTAL OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH THE FURNISHING,
# PERFORMANCE OR USE OF THIS MATERIAL WHETHER BASED ON WARRANTY, CONTRACT,
# OR OTHER LEGAL THEORY.
################################################################################

use IPC::Open2;

my $svc = shift @ARGV;
my $label = shift @ARGV;

# Connect to service engine, open streams

open2($IN, $OUT, "opcsvcterm");

$line = <$IN>;

if ($line =~ /xml/) { print "connected\n" } else { die "Cannot
connect to service engine"; }
```

```
    # Send XML to service engine
    print $OUT  "<Operations>\n";
    $line = <$IN>; die if !($line =~ /Results/) ;

    print $OUT    "<SetAttributes>\n";
    print $OUT      "<ServiceRef>$svc</ServiceRef>\n";
    print $OUT        "<Attribute>\n";
    print $OUT          "<Name>ov_label1</Name>\n";
    print $OUT          "<Value>$label</Value>\n";
    print $OUT        "</Attribute>\n";
    print $OUT    "</SetAttributes>\n";

    $line = <$IN>;
    if ($line =~ /OK/) { print "OK. Set label $label at service
 $svc\n" }
    else { die "Failed to set label $label at service $svc "; }

    print $OUT  "</Operations>\n";


exit(0);
```

For more information

For more information on HP OpenView Operations and HP Management Software, access the HP site at http://www.managementsoftware.hp.com

Call to action

To help us better understand and meet your needs for HP OpenView information, please send comments about this paper to: vesna.soraic@hp.com.

© 2005 Hewlett-Packard Development Company, L.P. The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

OV-ENXXXXXX, 01/2005