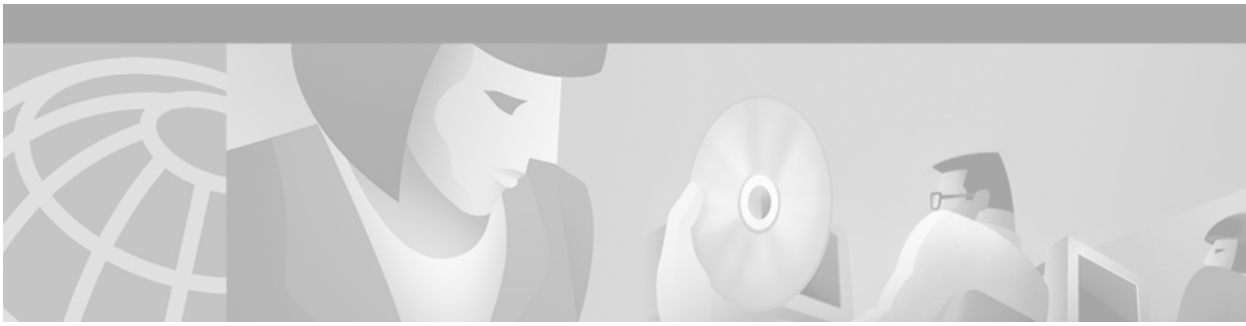


Task 7—Using HP OpenView to Create the SNMP Framework

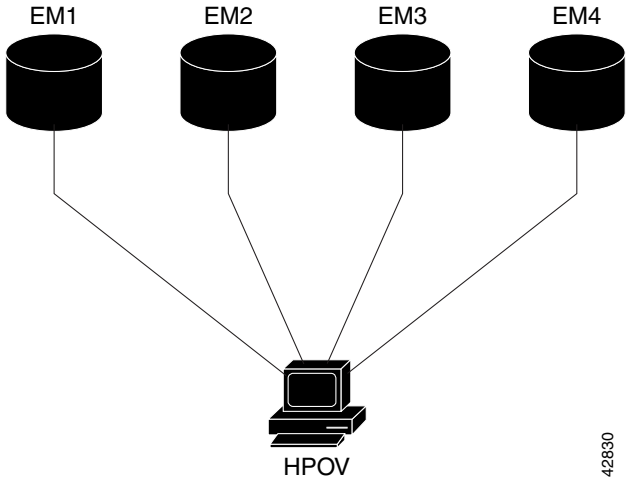
About HP OpenView

The primary function of HP OpenView (HPOV) is to manage faults.

In this case study, HP OpenView:

- Discovers all the devices in the network.
- Functions as the central-starting point for other element managers (EM). After HPOV is installed, the remaining components of the network management architecture are built around HPOV.
- Resides on the same Unix workstation as CiscoWorks 2000 Resource Manager Essentials, which gathers the following database information from HPOV:
  - Device names and IP addresses
  - Community strings

Figure 25 Other Element Managers Start from HPOV



Note This section assumes that HP Network Node Manager Release 5.0 is already installed on a Solaris workstation.

Describing the advanced capabilities of HPOV is outside the scope of this document. For more information, go to http://ovweb.external.hp.com/lpe/doc_serv/ and http://www.openview.hp.com

For Cisco IOS SNMP configurations, see the "Task 1—Enabling SNMP in a Cisco IOS Device" section on page 41.

Verifying the SNMP Configuration

To verify that the HPOV daemons are running and the SNMP configuration is correct, follow these steps:

Step 1 Start HPOV from the command line by entering the **ovw&** command from the /opt/OV/bin directory:

```
aurora:/opt/OV/bin ->ovw&
[1]     5079
```

Step 2 Verify that all the HPOV daemons are running by entering the **ovstatus** command from the root directory:

```
aurora:/ ->ovstatus
 object manager name: OVsPMD
 state:               RUNNING
 PID:                 430
 exit status:         -

 object manager name: ovwdb
 state:               RUNNING
 PID:                 431
 last message:        Initialization complete.
 exit status:         -

 object manager name: ovtrapd
 state:               RUNNING
 PID:                 433
 last message:        Initialization complete.
 exit status:         -

 object manager name: ovactiond
 state:               RUNNING
 PID:                 434
 last message:        Initialization complete.
 exit status:         -

 object manager name: pmd
 state:               RUNNING
 PID:                 432
 last message:        Initialization complete.
 exit status:         -

 object manager name: ovtopmd
 state:               RUNNING
 PID:                 435
 last message:        Connected to native database: "openview".
 exit status:         -

 object manager name: netmon
 state:               RUNNING
 PID:                 450
 last message:        Initialization complete.
 exit status:         -
```

```
object manager name: snmpCollect
state:               RUNNING
PID:                 451
last message:        No values configured for collection.
exit status:         -

object manager name: ovrepld
state:               RUNNING
PID:                 452
last message:        Initialization Complete.
exit status:         -
```

Note If a daemon is not running, try restarting it by using the commands **ovstop** *daemon-name* and **ovstart** *daemon-name*. If a daemon is still not running, an HPOV license issue may exist. For more information, go to http://www.openview.hp.com

Step 3 From HPOV, enter the SNMP community strings and target loopback IP addresses for each Cisco IOS device. From the **Options** menu, select **SNMP Configuration**.

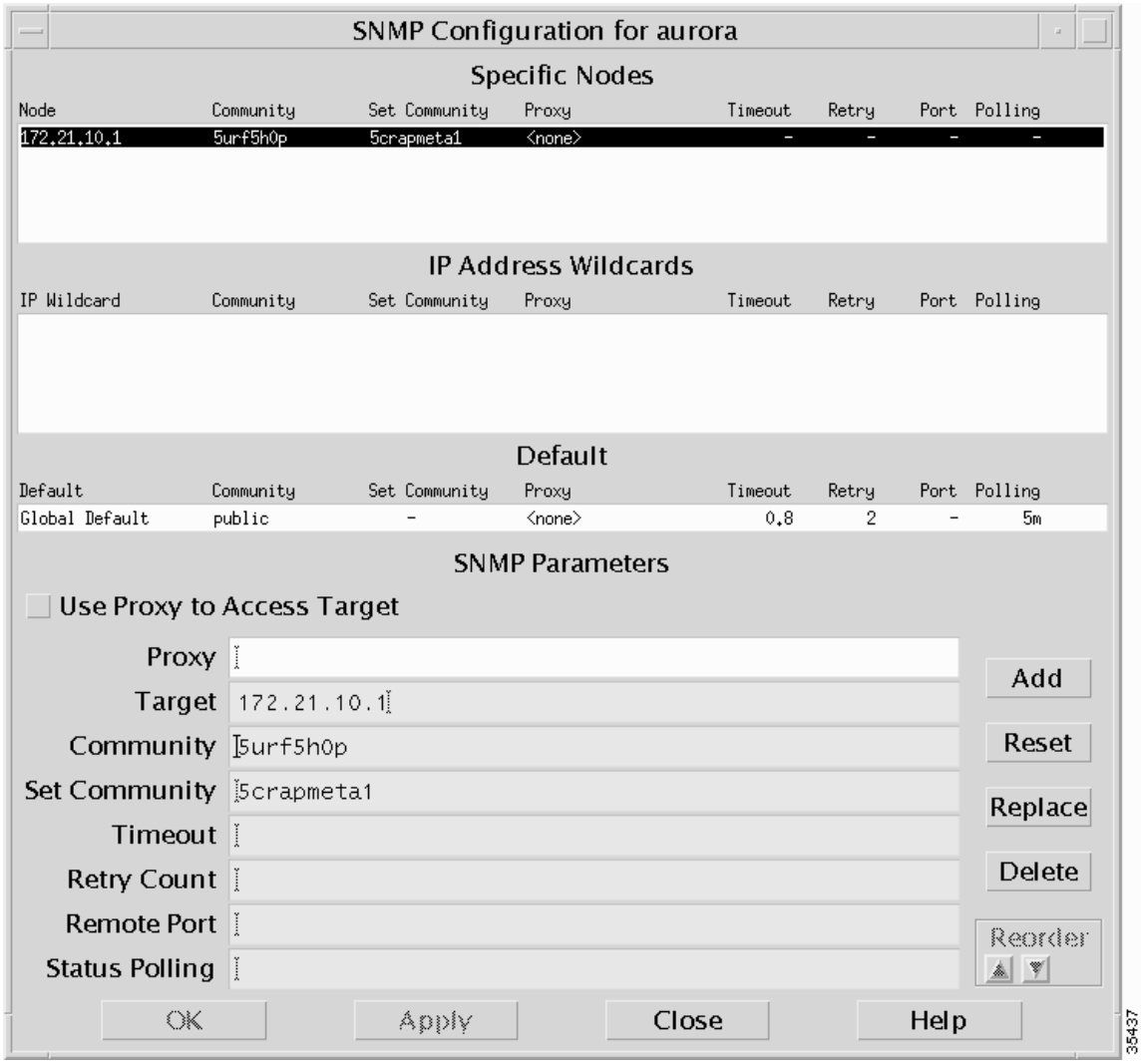
In the SNMP Configuration screen, enter the following information:

- Target field—The target loopback IP address (for example, 172.21.10.1)
- Community field—The Read-Only (RO) community string (for example, 5urf5h0p)
- Set Community field—The Read-Write (RW) community string (for example, 5crapmeta1)

Note Accept the default SNMP parameters in the other fields in the SNMP Configuration screen.

Caution Do not use the SNMP community strings “public,” “private,” or “cisco.” These strings are well-known within the industry, and they are common defaults. These strings are open invitations to attacks—even if you use filters.

Figure 26 SNMP Configuration: Loopback IP Address and Community Strings



Step 4 Click **Add** and **Apply** to submit the entries.

About SNMP Demand Polls

Perform an SNMP demand poll for a new managed device if you do not want to wait for the next automatic topology poll. HPOV performs less frequent automatic topology demand polls as your network and the HPOV device database becomes more static.

When the HPOV daemons start, HPOV discovers the devices in your network. Depending on which discovery options are configured, the device map is based on Layer 2 or Layer 3 information. Choosing discovery options is outside the scope of this document.

Depending on the number of devices that need to be discovered, it could take hours or even days for HPOV to discover a device. If HPOV cannot find a device, enter the device manually in to the database. See the “Using the HPOV CLI to Enter a Device into the Database” section on page 118.

To organize and adjust the top-level map, see the “Creating and Adjusting Maps” section on page 111.

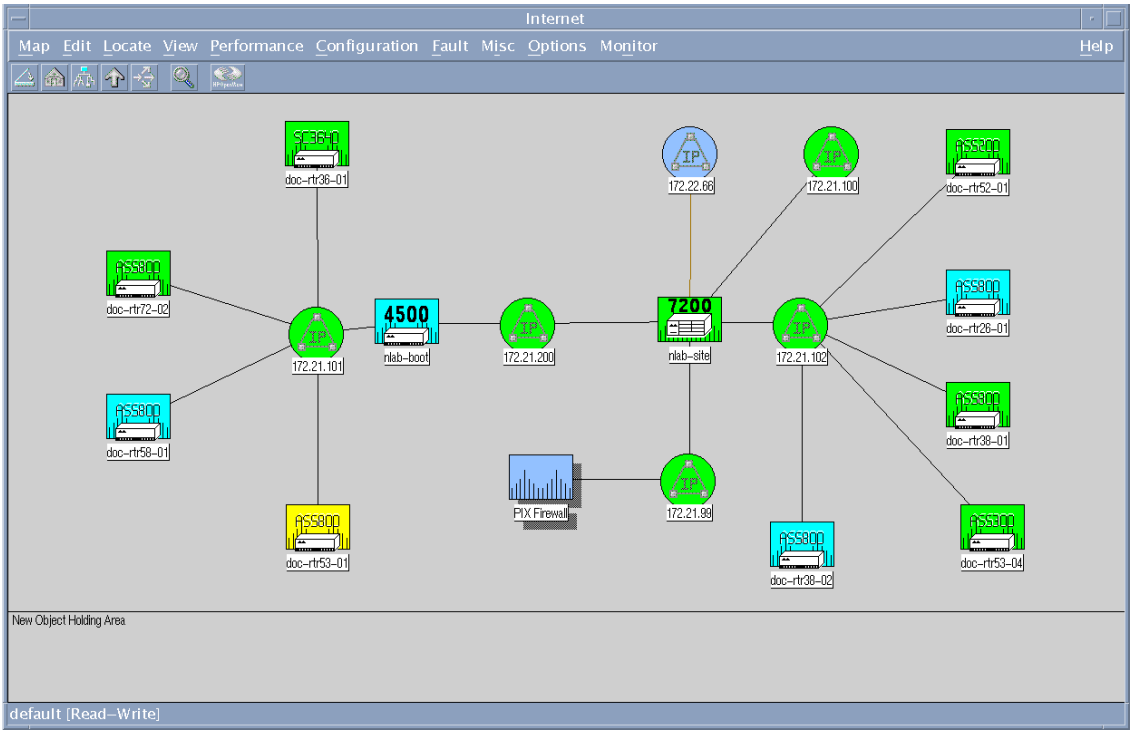
Performing an SNMP Demand Poll

To perform an SNMP demand poll, follow these steps:

Step 1 From the Root screen, double click the planet Earth Internet icon.

Step 2 Inspect the top-level map of the discovered devices in your network.

Figure 27 The Top-Level Device Map



Map color legend:

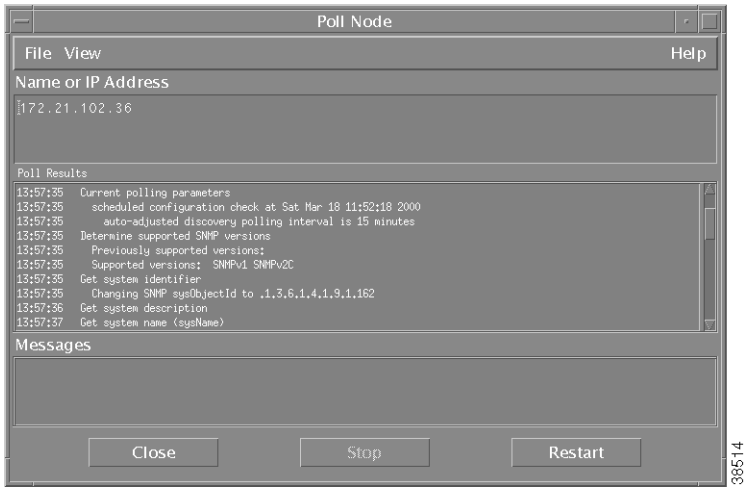
- Green—The device is up.
- Yellow—Multiple interfaces are down.
- Light blue—One interface is down.
- Dark blue—The device is detected, but it has never been managed. The device is unreachable.
- Red—The device is down and unreachable.

Step 3 Select a device icon in the map (single click).

Step 4 Go to **Fault**.

Step 5 Select **Network Connectivity: Poll Node**.

Figure 28 SNMP Walk-Polling Results



Demand polls enable HPOV to:

- Detect the sysobjectID (vendor ID) for each Cisco device.
- Associate MIBs with each device.
- Collect interface information.

Table 31 Important Fields to Inspect In the Polling Results

Field	Description
`Changing SNMP sysobjectID to .1.3.6.1.4.1.9.1.162`	Indicates SNMP is working and the system identifier for the device was found. This field appears only the first time a device is successfully polled. HPOV changes a generic router icon into a Cisco device icon after the sysobjectID is found. The trailing number series, for example .1.3.6.1.4.1.9.1.162, is the OID that identifies a node as a Cisco device.
`Supported versions`	Describes which versions of SNMP are supported by HPOV, such as SNMPv1 and SNMPv2C.
`Verify node name`	Verifies the node name is valid.

Table 31 Important Fields to Inspect In the Polling Results (continued)

Field	Description
`Interface`	Confirms the interfaces were successfully pinged.
`Get system description`	Verifies that the system description information was collected, so you can identify the software version running on the device.

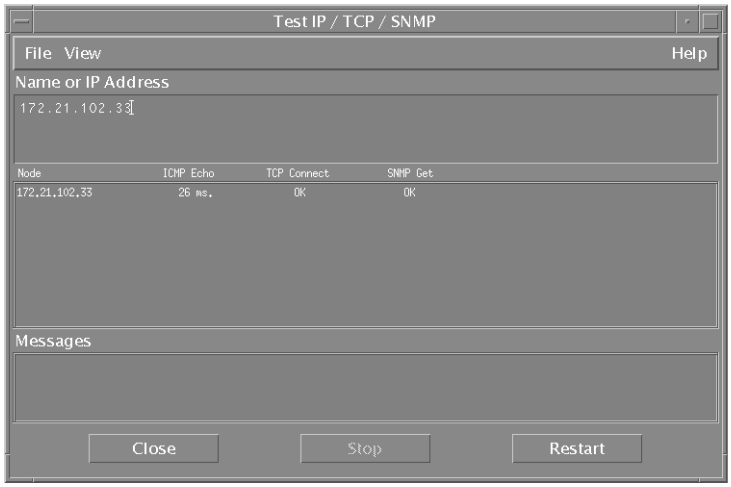
Testing SNMP Get Requests

To test that a device responds to SNMP **Get** requests, follow these steps:

Step 1 Select a device icon in the map (single click).

Step 2 From the **Fault** menu, select **Test IP/TCP/SNMP**.

Figure 29 Successful SNMP Test



This action performs one ICMP echo, one TCP connection, and one SNMP get. SNMP is working if the "OK" message appears under the SNMP Get field.

Table 32 describes the important fields in Figure 29.

Table 32 Test IP/TCP/SNMP Field Descriptions

Field	Returned Value	Description
Node	172.21.102.33	The target loopback IP address of the Cisco device.
ICMP Echo	26 ms	HPOV successfully pinged the device.
TCP Connect	OK	HPOV successfully made a TCP connection with the device.
SNMP Get	OK	HPOV successfully made an SNMP query to the device.

Troubleshooting SNMP and a Demand Poll

If a device is not responding to a demand poll, follow these steps:

Step 1 Poll a different device to see if it responds to SNMP. If the device responds, HPOV is not the problem.

Step 2 Ping the device that is not responding. If the ping works, the devices are communicating.

Note A firewall in the communication path can block ping and SNMP packets.

Step 3 Verify that the SNMP community strings are correct.

Step 4 Try polling the device from the HPOV command line. For example, enter the commands **snmpwalk** and **snmpget**.

The syntax for the **snmpget** command line is as follows:

```
snmpget [options] node object-id [object-id]...

Options:
     -d                 dump ASN.1 packet trace
     -v version         protocol version (1 or 2c)
     -c community       community string
     -p port            remote port
     -t timeout         retransmission timeout (1/10th seconds)
     -r retries         maximum retransmission attempts
```

Caution Overpolling the wrong OIDs overloads CPUs and crashes network devices.

Verifying that SNMP Traps Are Received

Traps appear in the All Events Browser, which reports what is happening in the network. The events are updated every few seconds. Understanding the severity level of a trap is important. One trap can be critical; whereas, another trap can be informative.

Other ways to look for traps include:

- Using a network analyzer to capture and inspect data on the line.
- Using the **snoop** Unix command to sniff the line and inspect data.

To monitor the limits of a network, configure thresholds to set off alarms. For example, set up an alarm for a CPU that sustains a 98 percent utilization for a specific amount of time.

Common mistakes include:

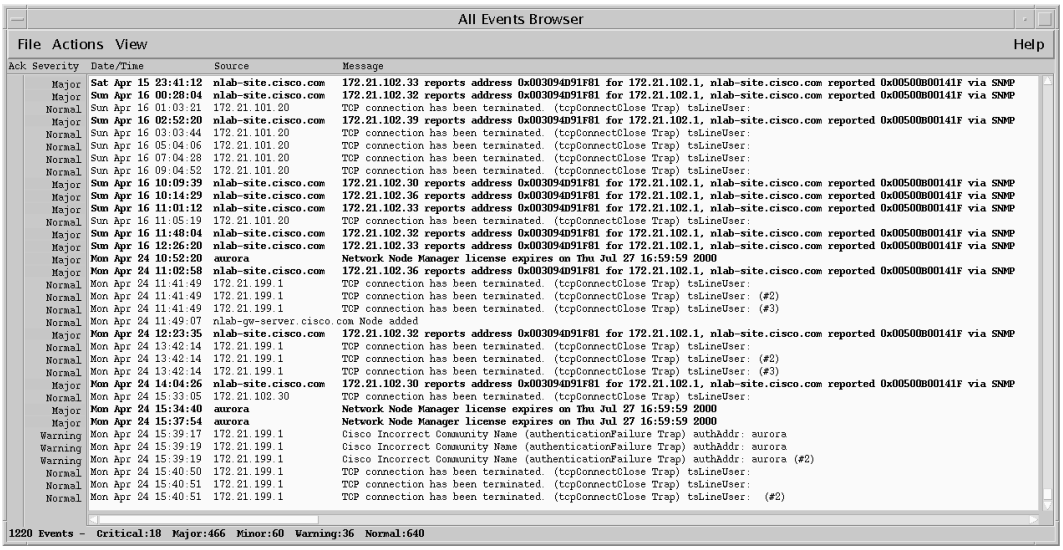
- Setting thresholds too low.
- HPOV is in a constant alarm state because you do not understand how to operate or monitor the dynamics of the equipment.

Setting up alarms for different kinds of traps is outside the scope of this document.

To verify that HPOV is receiving traps from devices in the network, follow these steps:

Step 1 Open the All Events Browser. From the **Fault** menu, select **Events**.

Figure 30 Traps in the All Events Browser



Step 2 Force a trap to be sent into the browser by manually causing a fault. Pull out a card on a Cisco device or shut down an interface.

Caution Do not shut down a communication link that can cause a service outage.

Step 3 Look for traps in the browser.

Unmanaging the Dial Ports

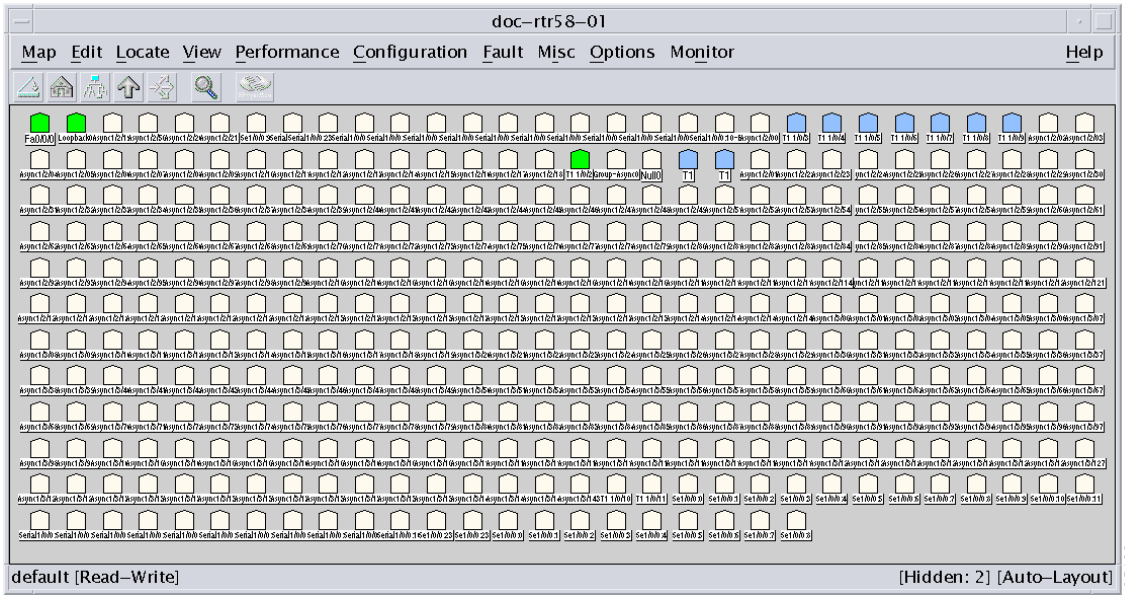
Do not poll the asynchronous and serial interfaces on Cisco access servers. The reasons for this recommendation include:

- As remote users dial in to an access server, it is normal behavior for asynchronous and serial interfaces to regularly go up and down.
- On an average 20-minute call, one modem normally produces three alerts. At approximately 6 alerts per hour, one modem can produce up 144 events each day. One Cisco AS5800 fitted with 1296 modems can produce up to 186,624 modem events per day.

To unmanage the asynchronous and serial interfaces for a Cisco access server, follow these steps:

Step 1 From the top-level map, double click on an access server icon. The available interfaces and ports appear.

Figure 31 Available Interfaces and Ports for a Cisco AS5800



Color legend:

- Green—The port is managed, and it is up.
- Blue—The port is managed, but it is administratively down on the Cisco IOS.
- Tan—The port is unmanaged.
- Red—The port is managed, but it is in a down state.

Step 2 Find the following interfaces:

- Serial interface channels (B and D channels). For example, Se1/0/0:6 and Se1/0/0:23
- Asynchronous interfaces. For example, Async 1/2/1

Step 3 Select a group of ports to unmanage. Draw a box around the ports, or select them individually.

Step 4 From the **Map** menu, select **Unmanage Objects**. Unmange all ports except the T1 trunks, loopback management interface, and Ethernet interface. Statistics are polled from managed ports.

Note You must unmanage the serial and asynchronous ports, which appear tan.

Tips When the status of an object changes (to managed or unmanaged), HPOV switches to synchronization mode.

Creating and Adjusting Maps

Maps provide a view of the network topology, and they enable you to quickly troubleshoot faults in the network. HPOV automatically polls devices and builds maps for you; however, devices often get stacked in the map, which is undesirable.

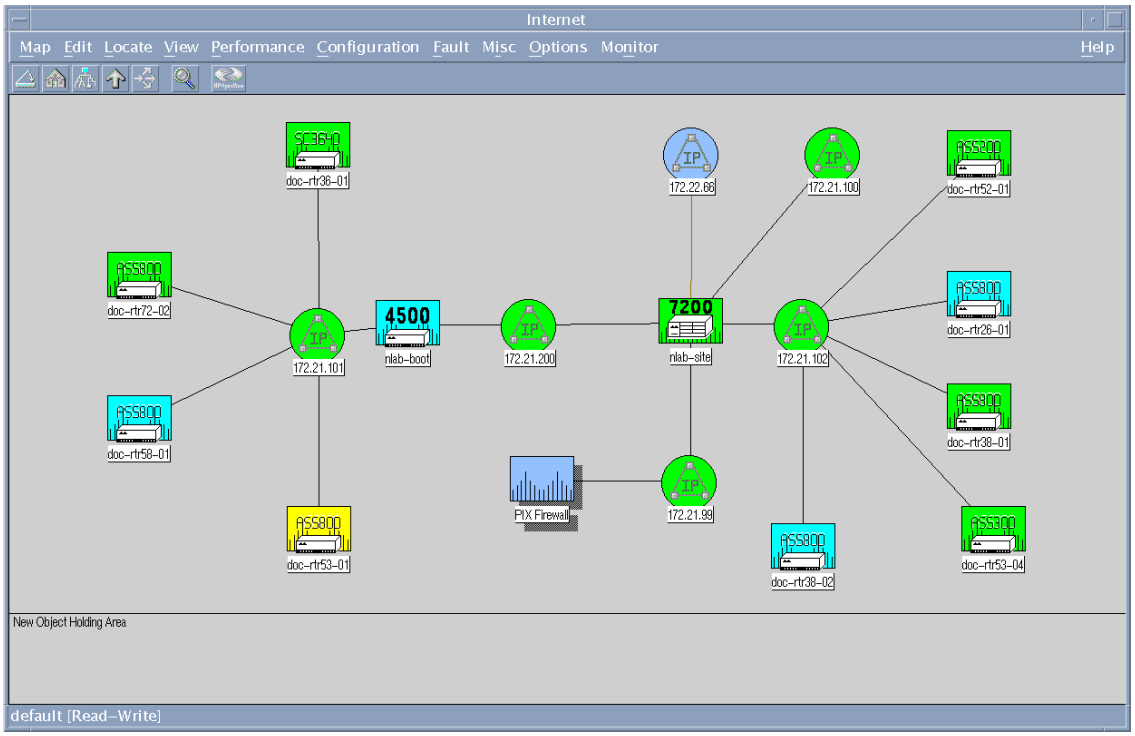
The following procedure saves you from having to refresh all your submaps each time a new device appears in the network. After you implement the following procedure, new devices will appear in the New Object Holding Area.

Caution Deleting a device from a submap removes the device from the database. To load a device back into the database, see the "Using the HPOV CLI to Enter a Device into the Database" section on page 118.

To manually re-structure device maps to adequately represent your network and turn off the automatic-layout function for the top-level map, follow these steps:

Step 1 Re-structure the top-level map by selecting and moving device icons. For example, put a collapsed backbone in the center of the map; then, position devices around the backbone.

Figure 32 Top-Level Map Adjustments



Step 2 Go to **View**.

Step 3 Select **Automatic Layout**.

Step 4 Choose **Off For This Submap**.

About Discovery Filters

A discovery filter is an ASCII file that HPOV reads to limit the discovery of devices on the network.

Use a discovery filter to:

- Define the subnets and devices you want to monitor.
- Avoid managing PCs and other non SNMP devices on the network.

Sometimes HPOV discovers too many devices. If HPOV discovers devices beyond your target network, such as the entire Internet, the performance of the Unix host decreases significantly. If the device maps begin filling up with networks, routers, and other devices that do not belong to you, use a discovery filter.

After a filter is set up, HPOV will not discover devices unless they are defined by the filter. Edit the filter each time a new device is added to the network.

For more information about discovery filters, go to http://www.openview.hp.com

Setting Up and Editing a Discovery Filter

The filter file is located in the /etc/opt/OV/share/conf/C directory. A sample file is shown in the following step-by-step example. The file has been manually edited and abbreviated to include a specific node list and filter list for this case study:

- **Node list**—A list of specific devices. In the example, the list is called TheNetNodes. There are two devices in the list: AS5800-1 and AS5800-2.
- **Filter list**—A list of attributes for the specified devices. In the example, the list is called TheNetFilters, which specifies the filtering attributes for the devices in the node list. For example, all devices must be SNMP compliant and Cisco devices.

The following are examples of complete filter files. These filters are examples which may be useful in your environment. Feel free to modify these filters or add your own.

The following example is the default filter file:

```
aurora:/etc/opt/OV/share/conf/C ->ls
filters     oid_to_sym  trapd.conf
aurora:/etc/opt/OV/share/conf/C ->cat filters
@(#)$OV_CONF/$LANG/filters
@(#)HP OpenView NNM Release B.05.01  Jun 21 1997
@(#)Copyright (c) 1990-1997 Hewlett-Packard Company
$Revision: /main/TORNADO/NNM_NT/5 $  $Date: 1997/01/13 19:35 UTC $
```

Note The behavior of topology filters in a distributed environment changed as of DFIX 5027. This file documents the behavior as of that patch level. This should be considered the correct specification of how topology filtering behaves in a distributed environment.

Sets are a simple way to list string values to test against in a filter. The IN operator tests a field value for membership in a set defined in the following example:

```
servers "Set of Servers" { "sv1", "sv2", "sv3" }
gateways "Backbone gateways " { "gw1", "gw2", "gw3" }
TheNetNodes "TheNet Node List" { "AS5800-1", AS5800-2" }
```

Filters and filter expressions are used as topology, discovery, map, or persistence filters. Applications that use filters (like ipmap(1), ovtopmd(1m), ovtopodump(1), and ovtopofix(1m)) can take the name of a filter or a filter expression.

The following filters are potentially useful as either discovery or topology filters, or components for building map filters.

Discovery Filters

Networks and segments are not directly testing with discovery filters, so there is no need to include isNetwork || isSegment in these filters. You can include networks or segments in a discovery filter, but they are not acted upon.

Because nodes are only discovered in managed networks, the network must already exist and be managed before the filtering test can be applied. New networks connected to discovered nodes can be discovered unmanaged. For discovery filters, a node and all its interfaces can be thought of as a single object. If any interface or the node itself passes the filter, the node and all of its interfaces will pass the filter.

The file called ovtopofix -f follows the discovery filter semantics and only applies to locally monitored objects. However, if a network or segment is empty after the processing of the nodes with the -f option, that segment or network is also removed. The ovtopodump(1) file also follows the discovery filter semantic, and the filter only applies to nodes and interfaces.

The following are discovery filter examples:

```
Networks "Any network" { isNetwork }
Segments "Any segment" { isSegment }
Nodes "Any node" { isNode }
```

Map and Topology Filters

For these filters, all objects are subject to filtering. You must specify the networks and segments to be passed. Furthermore, objects are displayed and included only if all their parent objects (submaps) pass the map and topology filter. For example, if no networks would pass the filter (isNetwork is not included), then all segments and non-gateways nodes would be filtered out. Like discovery filters, if any interface or node passes a map filter, the node and all of its interfaces also pass the filter.

See the Filter Expressions section for map and topology filter examples.

Note Prior to DFIX 5027 topology, filtering behaved similar to discovery filtering.

The following filters are primarily for example purposes, though they might be useful as discovery filters. They do not generally make sense as map or topology filters because they do not specify the inclusion of networks and segments. They also make sense as parts of a filter expression.

```
Routers "Any Router" { isRouter }
IPRouters "Any IP Router" { isIPRouter }
Bridges "Any bridge" { isBridge }
Hubs "Any multi-port repeater" { isHub }
SNMPNode "Any node supporting SNMP" { isNode && isSNMPSupported }
LocalLAN "Nodes with interfaces on the 15.2.112-119 subnet" { "IP Addre}
NeighborLANs "Node with interfaces on the neighboring LANs on site"
        { "IP Address" ~ 192.1.2-10.* }
HPNodes "Hewlett-Packard nodes"
        { isNode && ( vendor == "Hewlett-Packard" ) }
UBNodes "Ungermann-Bass Nodes"
        { isNode && vendor == "Ungermann-Bass" }
NonHPNodes "Non-HP nodes"
        { isNode && ( vendor != "Hewlett-Packard" ) }
```

The following example filters use the sets defined above:

```
GatewaysSet "Any designated Gateway node" { "IP Hostname" in gateways }
ServersSet "Any designated Server node" { "IP Hostname" in servers }
```

The next two filters are useful for defining map and topology filters (see the Filter Expressions section). They are not particularly useful as discovery filters because all networks and segments automatically pass discovery filters. Nor are they useful as standalone map filters, since they do not pass any nodes.

```
NetsNSegs "All networks & segments" { isNetwork || isSegment }
```

The network name below assumes an entry in /etc/networks. If no such entry exists, the actual network name can be something like “15.2.112.”

```
MyNet "The network in which I'm interested"
        { isNetwork && "IP Network Name" == "yellow-lan" }
```

The next filter example shows a way to pass all objects contained in the subnet 15.2.112.0 with a subnet mask of 255.255.248.0 (so the subnet range is 15.2.112.0 to 15.2.119.255). The inclusion of isSegment is not a concern, because all segments outside of this network would be filtered out because the networks containing them were filtered out. You must have some specification of segments if you want the segment contents to be included.

```
EngrLan "The 15.2.112 subnet used by engineering"
        { ("IP Address" ~ 15.2.112-119.* ) || isSegment }
```

The following filter accomplishes the same thing as the above example but is more restrictive in the specification of segments to only include segments in that engineering network. This assumes there is an entry in /etc/networks to make sure that the subnet gets the name engr-LAN, and the segments in that network all have a name of form engr-LAN.<variable>. Generally, the filter in the previous example can accomplish most tasks, but using the following example might be necessary in some situations if you want specific segments. In particular, it works well if you want to use this to EXCLUDE the objects in this network. See the example in the Filter Expressions section below.

```
EngrLan2 "The 15.2.112 subnet used by engineering"
        { ( "IP Address" ~ 15.2.112-119.* ) ||
          ( "IP Segment Name" ~ "engr-LAN.*" ) }
```

The following filter example specifies that only objects that support IP should be included. This filter should be used as a topology filter on any collection station discovering non-IP information (Level 2 or IPX only nodes or networks) that is forwarding to an NNM 4.X management station. This filter does not completely remove all non-IP information (for example, IPX interfaces on a node that also supports IP do come through), but it does limit the impact on the management station and make migration to NNM 5.X easier.

```
IPOnlyObjects "Only objects that support IP"
        { isIP || isSegment }
```

Filter Expressions

Filter expressions are simply combinations of filters defined in the same filter file (as above). Filter expressions make it simple to combine filters without reproducing the expressions of each filter again.

The following example combines the two set filters defined above into one filter expression. It works unmodified as a discovery filter. To make it work as a map filter, network and segment filtering must be added (see below).

```
VitalNodes "All Gateways and Servers" { GatewaysSet || ServersSet }
```

You can turn the filters defined above into viable map or topology filters by simply adding the following input:

```
|| NetsNSegs
```

Adding this input does not invalidate the filters as discovery filters; it just adds a superfluous test.

```
VitalNodesMap "All nets & segs, but only gateway and server nodes"
                { GatewaysSet || ServersSet || NetsNSegs}
        LocalLANView "All nets & segs, but only local nodes"
                { LocalLAN || NetsNSegs }
        NetInfrastructure "Any network connecting device and what they connect"
                { Routers || Bridges || Hubs || NetsNSegs }
        NetBackbone "Networks and gateways/routers"
                { Routers || Networks }
```

Using the filters defined above that include only a specific network, you can also exclude the specific network. Note the use of the more specific form to exclude only the segments in the engineering LAN. This could have been specified directly as a negation in the filter portion, but this form works well if you have several networks to manipulate in this manner.

```
        EverythingButEngr "Everything but the engineering LAN"
                { !EngrLan2 }
```

In the filter expressions above, when used as map filters, you can pass all networks and segments. You might want to see only a particular network. The following map filters accomplish this.

Note Though segments and nodes from other networks will pass the filters, IP Map will ignore them because their parent networks will not pass.

Note These filters will not work as discovery filters because all network and segments automatically pass discovery and topology filters.

```
        MyNetMap "Only the network of interest and all its constituent parts"
                { MyNet || Segments || Nodes}
        MyVitalNodesMap "Gateways, servers and segments in the net of interest"
                { MyNet || Segments ||  GatewaysSet || ServersSet }
        TheNetNodeList "The this the filters for TheNet nodeslist
                { TheNetNodes || TheNetFilters }
```

The following example shows a map persistence filter, which ensures that all Ungermann-Bass are kept in memory and up to date. This will also keep any containing submaps in memory.

```
        PersFilter "Objects to keep in map memory" { UBNodes }
```

To set up and edit a discovery filter, follow these steps:

Step 1 Find the filters file on your Unix workstation:

```
aurora:/etc/opt/OV/share/conf/C ->ls
filters     oid_to_sym  trapd.conf
```

Step 2 Edit the filters file by using a text editor to include a node list and a filter list for your network environment:

```
aurora:/etc/opt/OV/share/conf/C ->vi filters
//
// @(#)$OV_CONF/$LANG/filters
// @(#)HP OpenView NNM Release B.05.01  Jun 21 1997
// @(#)Copyright (c) 1990-1997 Hewlett-Packard Company
// $Revision: /main/TORNADO/NNM_NT/5 $  $Date: 1997/01/13 19:35 UTC $
//
```

```
// This is the default filter file.  These filters are examples
// which may be useful in your environment.  Feel free to modify
// these filters and/or add your own.  See OVfilterIntro(5)
// for more information on this file.

// NOTE: The behavior of topology filters in a distributed environment
// changed as of DFIX 5027.  This file documents the behavior as of that
// patch level.  This should be considered the correct specification of
// how topology filtering behaves in a distributed environment.

//
// Sets are a simple way to list string values to test
// against in a filter.  The "IN" operator tests a field value
// for membership in a set defined here.
//
Sets {
//
// These are simple examples of sets.
//
servers "Set of Servers" { "sv1", "sv2", "sv3" }
gateways "Backbone gateways " { "gw1", "gw2", "gw3" }
TheNetNodes "TheNet Node List" { "AS5800-1", "AS5800-2" }

}
.
.
.
FilterExpressions {
        //
        // The following combines the two set filters
        // defined above into one FilterExpression.
        // It works unmodified as a discovery filter.
        // To work as a map filter, network and segment filtering
        // must be added (see below).
        VitalNodes "All Gateways and Servers" { GatewaysSet || ServersSet }

        //
        // One can turn the filters defined above into viable map or
        // topology filters by simply adding "|| NetsNSegs".  (Doing so
        // does not invalidate the filters as discovery
        // filters.  It just adds a superfluous test.)
        //
        VitalNodesMap "All nets & segs, but only gateway and server nodes"
                { GatewaysSet || ServersSet || NetsNSegs}
        LocalLANView "All nets & segs, but only local nodes"
                { LocalLAN || NetsNSegs }
        NetInfrastructure "Any network connecting device and what they connect"
                { Routers || Bridges || Hubs || NetsNSegs }
        NetBackbone "Networks and gateways/routers"
                { Routers || Networks }

        // Using the filters defined above that include only a specific
        // network, we can also exclude the specific network like this
        // Note the use of the more specific form to exclude only the segments
        // in the engineering lan.  This could have been specified directly
        // as a negation in the filter part, but this form works well if you
        // have several networks to manipulate in this manner.
        EverythingButEngr "Everything but the engineering LAN"
                { !EngrLan2 }

        // Of course the above filter expressions, when used as
        // map filters, pass all networks and segments.  You
        // may wish to see only a particular network.  The following map
        // filters accomplish this.  Note that though segments
```

```
        // and nodes from other networks will pass the filters, IP Map
        // will ignore them because their parent networks will not pass.
        // NOTE:  These filters will not work as Discovery
        // filters because all network and segments automatically pass
        // Discovery and Topology filters.
        //
        MyNetMap "Only the network of interest and all its constituent parts"
                { MyNet || Segments || Nodes}
        MyVitalNodesMap "Gateways, servers and segments in the net of interest"
                { MyNet || Segments ||  GatewaysSet || ServersSet }
        TheNetNodeList "This is the filter for TheNet nodeslist
                { TheNetNodes || TheNetFilters }

        // This is a map persistence filter which ensures that
        // all Ungermann-Bass are kept in memory and up to date.
        // Note that this will also keep any containing submaps in memory.
        //
        PersFilter "Objects to keep in map memory" { UBNodes }
}
```

Using the HPOV CLI to Enter a Device into the Database

Sometimes devices do not appear in the device map, or they are accidentally deleted from the HPOV database.

To manually load devices in to the HPOV database by using the CLI, follow these steps:

Step 1 This step ensures that new host entries are safely loaded in to the database. Shutdown the netmon daemon by entering the **ovstop netmon** command from the root directory. All automatic network polling and database updates stops.

```
aurora:/ ->ovstop netmon
aurora:/ ->ovstatus netmon
 object manager name: netmon
 state:               NOT_RUNNING
 PID:                 450
 last message:        Exited due to user request
 exit status:         Exit(0)
```

Step 2 To load new devices in to the database, enter the **loadhosts -m** command from the root directory followed by a single netmask for the devices. Include an end of file statement (EOF) to enter multiple lines with one return.

```
aurora:/ ->loadhosts -m 255.255.255.0 <<EOF
> 10.10.10.104     hostname
> 14.14.14.14       host2name
> EOF
aurora:/ ->
```

Note Enter devices by using a DNS format (IP address then hostname). Use spaces (not tabs) to separate IP addresses from hostnames.

Step 3 Restart the netmon daemon by entering the following commands:

```
aurora:/ ->ovstart netmon
aurora:/ ->ovstatus netmon
```

```
object manager name: netmon
state:               RUNNING
PID:                 12812
last message:        Initialization complete.
exit status:
```

Step 4 Go to the GUI and look for the new devices that appear in the new object holding area.

Step 5 Perform a demand poll on each device to get the sysobjectIDs. After the demand poll is performed, HPOV puts each new device into its correct place in the map.