Overview of Operation

History

netmon started as “discover,” a process which polled the ARP caches of the nodes on a network to find out all the nodes on that network. “discover” was deemed too restrictive because it was a synchronous polling process:

- discover was spending most of its time waiting for non-SNMP nodes to time out.
- discover did not idle; it would spin when there was nothing to do.
- discover did not receive events (traps).

Therefore, the main goal of netmon was to create an asynchronous SNMP and ICMP polling machine. This goal has driven most of the design.

Terminology

- The words “event” and “trap” are used interchangeably throughout this document. They are received via the pmd process.

- “ICMP polling” refers to polling by use of ICMP Echo requests, or what are commonly known as “pings.” For nodes which do not support SNMP, the ICMP Network Mask request is occasionally substituted for the Echo request. Certain nodes do not respond correctly to the Network Mask request. netmon detects this and does not make the substitution for subsequent polls.

- A “node” is a machine on the network; that is, a computer, a router, a hub, a bridge, etc. A node usually contains one or more interfaces. Starting at NNM 2.0, netmon supports bridges and hubs from HP’s Workgroup Networks Division (in Roseville, California, formerly known as RND), originally referred to as ICF devices. Support has been extended to include the AdvanceStack and VG-anyLAN devices as well.

- An “interface” is an object which attaches a node to a particular segment and/or network. Usually, it has an IP address and a link level address. netmon attaches bridges and hubs using extra interfaces which usually do not have any address information, but they will have a port or port group. for these purposes. The interfaces with IP addresses on bridges and hubs are contained in their respective network, but are not connected to any segment in that network.
Overview of Operation

External Behavior

What does netmon do? The vision for netmon is some magic in the background which knows all (at least about networks) and reports problems and failures when they occur. What does netmon actually do?

- periodically does ICMP polling (pings and mask requests) and SNMP polling.
- can have multiple ICMP and SNMP polls outstanding.
- responds to events (traps) generated by nodes on the network and by other components of the system.

ICMP Polling

netmon performs ICMP polling for all managed interfaces in the topology database. The interval between polls is determined on a per-address basis, as configured via the SNMP configuration dialog.

**NOTE:** netmon performs ICMP polling for interfaces, not nodes.

netmon sends down and up events for interfaces and performs status propagation. That is, it computes the combined status of nodes, segments, and networks, sending additional events as necessary. netmon generates an interface up event when it receives any ICMP echo from an interface that was marked as down in the database. If the interface is marked as up in the database but is down in actuality, netmon will transmit ping packets for the configured number of retries before it decides that the interface is down.

By default, netmon can have up to three ICMP requests outstanding (packets sent without having a response). This number was originally chosen to match the length of the kernel queue reserved for the ICMP socket. Although this restriction on the ICMP socket no longer exists, keeping the number of parallel requests under control does reduce the amount of LAN bandwidth and CPU time consumed by the status polling activity.

**NOTE:** All ICMP activity comes through the single kernel-resident ICMP socket, thus netmon might lose some packets if there are other processes using ICMP. The lost packets will be resent. netmon will also receive ICMP responses to requests that it did not generate. These “unexpected responses” are a source of discovery hints.

netmon also sends out ICMP echo requests to all the “hint” IP addresses. “Hints” are IP addresses which are found through other nodes (most often through the ARP cache or the routing table). Since the “hints” could come from questionable sources, netmon verifies the interface’s existence by pinging it. If an interface responds to a ping request, a node object is created for it and it is added to the database after performing an Initial Configuration operation (See the section on Predefined SNMP polls below). Hints are not discarded after one failure; netmon will try 20 consecutive times before giving up and “forgetting” the hint. The interval between attempts starts with the
configured polling interval for that address, obtained via the OVsnmp configuration API, and gradually checks less frequently with each failed attempt.

After a node is initially discovered, an ICMP Mask Request is sent instead of the next scheduled Echo Request. If a reply is received, the returned mask is compared against the expected netmask for the network, and an event is generated if they do not match. If no reply is received, a ping is sent to verify that the node is alive. If a reply to that ping is received, no Mask Request will ever again be generated for this node (since the node clearly doesn't support the request). Otherwise, the check is repeated regularly at the configuration checking interval. If the node is later found to support SNMP, the check is permanently disabled.

Predefined SNMP Polls

netmon requests and processes many different types of SNMP data. There are four predefined SNMP operations:

- The Initial Configuration operation is performed after a hint has been confirmed, i.e. after the first ping of a new IP address succeeds.
- The Daily Configuration Check operation is performed once each day (by default, this period is configurable) on each node; the idea is to see if any relevant configuration information has changed since the last time we checked.
- The Discovery operation, also referred to as an AT Check operation (after the Address Translation table of MIB-I). This operation checks the node for information which might lead netmon to other nodes, i.e. other hints.
- The Demand Poll operation is performed in response to a request from the user for a poll of a particular node. This poll is a combination of the Configuration and Discovery polls.

Based on the information it receives via SNMP, netmon will:

- look for hints of previously unknown nodes, and
- store the information in the topology database so that it may be:
  - compared with past values to see if "change" events should be generated.
  - retrieved by other applications, particularly ipmap.

Configuration Check Operation

The following is a list of the SNMP requests generated by netmon during both the Initial and Daily Configuration Check operations and a description of what is done with the information:

- system object identifier (system.sysObjectID)
  - stored in database
  - generates event if changed
- used to identify nodes which are treated specially (see Object ID Mapping)

• system descriptor \texttt{(system.sysDescr)}
  - stored in database
  - shown in ipmap's attribute box
  - generates event if changed

• IP address table \texttt{(ip.ipAddrTable)}
  - fields \texttt{ipAdEntNetAddr}, \texttt{ipAdEntNetMask}, and \texttt{ipAdEntIfIndex} are retrieved
  - stored in the database
  - entries \texttt{ipAdEntNetAddr} and \texttt{ipAdEntNetMask} are shown in ipmap's attributes box
  - netmon gets the \texttt{ipAdEntNetAddr} for the IP address; this could possibly result in a new interface event.
  - netmon gets \texttt{ipAdEntIfIndex} for matching the IP address with the interface table's entry
  - netmon gets the \texttt{ipAdEntNetMask}. If the IP address is the first on the network, then this becomes the network mask for all subsequent interfaces on the same network. Otherwise it is compared to the current netmask for that network. If it doesn't match, an event is generated.

• Interface number \texttt{(interfaces.ifNumber)}
  - used to limit the number of rows to be requested (to protect netmon from defective agents)

• Interface table \texttt{(interfaces.ifTable)}
  - fields \texttt{ifIndex, ifPhysAddress, ifType, ifOperStatus, ifAdminStatus, and ifDescr} are retrieved.
  - stored in the database
  - the fields are visible in ipmap's attribute box
  - the \texttt{ifAdminStatus} field can change an interface's managed state
  - \texttt{ifDescr, ifPhysAddress, and ifType} can cause change events.
  - \texttt{ifType} helps determine the segment type to which the interface will attach the node

• Interface table extensions
  - field \texttt{ifName}
  - stored in the database
  - used to name the interface
  - can cause a change event
• IP forwarding status (ip.ipForwarding)
  - stored in database
  - generates event if changed
  - helps determine topological role (router or host)

• system location (system.sysLocation)
  - stored in database
  - generates event if changed
  - shown in ipmap's attributes box

• system contact (system.sysContact)
  - stored in database
  - generates event if changed
  - shown in ipmap's attributes box

• Default route (ipRouteNextHop.0.0.0.0)
  - generates event if default route does not have routing characteristics
  - used for discovery hints

• Routing table (ip.ipRouteTable)
  - only retrieved if node has a WAN interface
  - only ipRouteNextHop retrieved
  - used for discovery hints on WAN links (which typically won't have corresponding ARP cache entries)

The above information is retrieved initially when the node is found (after it first responds to a ping) and polled for regularly at the configuration checking interval. No SNMP-related events are generated on the initial poll (since a new node event will be generated anyway). If any of the requests times out, then all the subsequent requests are aborted.

The isSNMP attribute is set based on whether the node has responded to any SNMP. If the node does not respond initially (to system.sysObjectID) after the configured number of retries, the node is marked as not supporting SNMP. However, netmon will try every configuration checking interval just in case SNMP becomes supported.

There are two set operations which are only done to nodes with HP object IDs (system.sysObjectID). The first one sets the agent capabilities to respond to further HP requests. This operation is done early on in the configuration poll. The second operation adds the management station to the trapDestination table so SNMP traps will be sent to the management station. This operation is done toward the end of the poll.

For HP hubs and bridges, there are additional topology related queries performed at the end of the configuration poll. These will be discussed in more detail in the Subnet Layout section below.
Discovery Operation

If a node supports SNMP, then a “new node discovery” operation is scheduled for it. New node discovery consists of the SNMP polling of the `ipNetToMedia` table (or if that fails, the deprecated Address Translation table), the default routing entry, and, for nodes having WAN interfaces, the entire routing table (but only for the first Discovery Operation done on a node).

- default routing entry (`ipRouteNextHop.0.0.0.0`)
  - used for discovery hints
- IP Net-to-Media Table (`ipNetToMediaTable`) or Address Translation table (`atTable`)
  - `ipNetToMediaTable` tried first (`atTable` retrieved only if `ipNetToMediaTable` not supported)
  - The objects `NetAddress` and `PhysAddress` of the `ipNetToMediaTable` (or `atTable`), the ARP cache for Ethernet interfaces, are retrieved. All new `NetAddresses` will be added to the list of hints if they are not already in the database. If the `NetAddress`'s link level address is already known, the link address associated with it will be compared to `PhysAddress`. If they don’t match, a node configuration event will be generated. If no link address is associated with the `NetAddress` in the database or the existing link address information did not come from the node itself, the database will be updated with the current `PhysAddress`.

The “new node discovery” polling interval is either fixed or dynamic. The dynamic (auto-adjusting) polling interval is provided so that traffic related to discovery decreases after most of the nodes on a network have been found. The auto-adjusting feature is applied on a per node basis. Whenever a node is polled, netmon determines how much “new” information it got from that node. If it receives nothing new, the discovery period is increased by a constant (5 minutes). If it receives new information for more than 5 nodes, the discovery period is halved. The minimum discovery polling period is 1 minute and the maximum is one day. These numbers are not configurable.

The user can specify a discovery filter via the Network Polling dialog which constrains the nodes discovered by netmon. See the Filtering Discovery section below for more details.

Demand Poll Operation

Demand polling is a way for the user to tell netmon to “do its thing” without waiting for the polling period. This is useful to update a node (if the user knows that the node has changed) and to verify that netmon is alive. The demand poll operation makes the same SNMP requests as the combination of the configuration and the discovery polls.

Demand polling from the user interface executes `nmdemandpoll` with a list of node names. All output of `nmdemandpoll` goes to the Demand polling dialog box. The following activities occur in the `nmdemandpoll` and netmon processes for each node during a demand poll:
1. *nmdemandpoll* sets up a TCP socket to the system which is the primary management station for the node.

2. *nmdemandpoll* sends a trap to *netmon* containing the node to be polled and the connection information (host and socket port number).

3. *netmon* receives the trap and:
   a. opens the socket.
   b. marks the node (in its memory resident topology) as being “demand polled.”
   c. schedules pings of the node’s interfaces one at a time and a demand poll operation on the node itself.
   d. prints out polling results to the socket.
   e. closes the socket once the polling is over.

6. *nmdemandpoll* reads from the socket and copies it to stdout.

**Custom Operations**

A custom operation is performed by *netmon* to get subnet topology information from bridges and hubs. These are not scheduled to occur on a regular basis but rather only as needed.

**Trap Handling**

*netmon* does some interpretation of SNMP traps. If the sending node doesn’t exist, *netmon* adds its address as a hint. Otherwise, external traps are handled as follows:

- **coldStart** and **warmStart**—Schedule an immediate configuration poll.
- **linkDown**—Schedule an immediate status poll for the address indicated in the trap.
- **linkUp**—If the sending node only had one IP address known to us, declare the node UP. Otherwise, schedule an immediate status poll for the address indicated in the trap.

Other trap are ignored.
Pertinent Internal Algorithms

netmon’s main loop

As stated before, the driving structure behind netmon is its asynchronous polling machine. netmon’s main loop consists of a `select()` system call waiting for one or more of the following:

- SNMP response
- ICMP response
- event from pmd
- Emanate master agent messages
- OVsPMD message
- timeout

The timeout of the main select is calculated as the minimum of the shortest timeout for a ICMP and SNMP responses and the time until the next scheduled query (if there are any slots available in the "wait" lists).

NOTE: The time used in netmon is obtained via the `time()` system call, which has a resolution of 1 second. It was felt that the simplicity of dealing with one integer quantity was worth the slight inaccuracy of the rough timing.

Lists

To track the communication process, netmon uses lists extensively. The routines surrounding the main loop use the following lists:

- **pingList**—A time ordered list containing all interfaces which need to have ICMP requests sent sometime in the future. All managed interfaces are in this list except when there is an ICMP poll in progress for an interface. Hints can also be on this list.
- **pingWait**—A time ordered list containing interfaces which are waiting for an ICMP response.
- **pausedPingList**—This is a list of interfaces which just received a ping response, and are waiting for the SNMP queries for a newly discovered node to complete or timeout. This is a temporary list and should be empty most of the time.
- **snmpList**—A time ordered list containing all nodes which need to have SNMP requests sent sometime in the future. All managed nodes are in this list except when there is an SNMP operation in progress for a node.
- **snmpWait**—A time ordered list containing nodes which are waiting for an SNMP response.
- **unSnmpList**—This is a list of non-active (not managed) nodes. The corresponding unmanaged interfaces should be on the network interfaces list for each of the networks with which the interfaces are associated.
- **network interfaces** (one per network)—If the network is managed,
this list contains interfaces which are unmanaged (by the node or
interface being unmanaged by the user). If the network is unmanaged, it
contains all interfaces known for that subnet, including hints of new
nodes.

NOTE: An interface will be on at most one of the ICMP polling lists at a time.
Similarly, a node will be on at most one of the SNMP polling lists at a time even
though there may be more than SNMP operation scheduled for a it

All interfaces which need to be pinged are put in the pingList based on the
absolute time when they need to be pinged. When the time comes, it is
removed from pingList, the request is sent, and the interface is put on the
pingWait list. When a response comes, the response is matched with the
correct request in the pingWait list. This interface is then moved back to the
pingList. If no response comes before the timeout configured for the
interface, it is left on the pingWait list, the timeout is set to the configured
number of seconds, and the retry timeout count is incremented by one. On
subsequent retries, the timeout period is doubled and the new timeout period
kept with the interface. This means that for an interface configured with a 1
second timeout and 3 retries, the first ping will timeout after 1 second, the
first retry after 2, the second retry after 4, and the third retry after 8 seconds.
After all retries have been exhausted, the interface is returned to pingList
and placed in the list based on the time of its next ping.

The handling of the SNMP lists is analogous to the ping lists, except there are
multiple SNMP “operations” scheduled for single node. Each node is entered
on the snmpList with the time for the earliest operation. Each operation
could involve many SNMP requests. A node in the snmpWait list is not
removed after receiving the first response of a sequence. It stays in the
snmpWait list until the last of the requests has been satisfied or the remote
node doesn’t respond to a request.

How busy is netmon?

netmon can report several statistics which indicate how well it is able to keep
up with the configured ICMP and SNMP polling activities. This is done via an
Emanage Subagent that is incorporated into netmon. netmon supports the
following MIB objects that relate to its polling:

HP-OV-NETMON DEFINITIONS ::= BEGIN
IMPORTS
   OBJECT-TYPE FROM RFC-1212;

hp OBJECT IDENTIFIER ::= { enterprises 11 }
nm OBJECT IDENTIFIER ::= { hp 2 }
onView OBJECT IDENTIFIER ::= { nm 17 }
hpOVDistibStation OBJECT IDENTIFIER ::= { openView 4 }
hpOVNetmon OBJECT IDENTIFIER ::= { hpOVDistibStation 4 }
nmStatus OBJECT IDENTIFIER ::= { hpOVNetmon 1 }
nmICMPStatus OBJECT IDENTIFIER ::= { nmStatus 1 }
nmSNMPStatus OBJECT IDENTIFIER ::= { nmStatus 2 }

-- ICMP polling objects
nmICMPLength OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-only
STATUS mandatory
DESCRIPTION "The length of netmon’s ICMP polling list, indicating the number of managed interfaces actively being status polled."
 ::= { nmICMPStatus 1 }

nmICMPPollsWithin1m OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-only
STATUS mandatory
DESCRIPTION "The number of interfaces scheduled to be polled by netmon via ICMP with the next minute."
 ::= { nmICMPStatus 2 }

nmICMPSecsUntilNextPoll OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-only
STATUS mandatory
DESCRIPTION "The number of seconds until the next scheduled ICMP poll. This may a negative number if netmon is behind in ICMP polling."
 ::= { nmICMPStatus 3 }

-- SNMP polling objects

nmSNMPLength OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-only
STATUS mandatory
DESCRIPTION "The length of netmon’s SNMP polling list, indicating the number of managed nodes actively being polled."
 ::= { nmSNMPStatus 1 }

nmSNMPPollsWithin1m OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-only
STATUS mandatory
DESCRIPTION "The number of nodes scheduled to be polled by netmon via SNMP with the next minute."
 ::= { nmSNMPStatus 2 }

nmSNMPsecsUntilNextPoll OBJECT-TYPE
SYNTAX INTEGER
ACCESS read-only
STATUS mandatory
DESCRIPTION "The number of seconds netmon until the next scheduled SNMP poll. It is calculated by subtracting the time of the poll at the head of the list from the current time. If netmon
is behind, this can be a negative value.”

::= { nmSNMPStatus 3 }

END

These MIB objects can be accessed in the form of a graph via “Performance -> Network Polling Statistics”.

There are also netmon command-line options to print the contents of each of its lists. See the Command Line Options section below.

**Filtering Discovery**

It is possible to restrict what netmon will discover by specifying a filter name in the Network Polling Configuration dialog. The discovery filter is applied with the following semantics:

- A node and its interfaces will be checked to see if it passes this filter just prior to adding the node to the database.
- A node object will pass the filter if any of its interfaces pass.
- An interface object will pass the filter if its containing node object or any of the other interface objects associated with node object pass.

If a node and its interfaces do not pass the discovery filter, the IP addresses of the node are added to a temporary “don’t discover” list and records the last time the nodes did not pass the filter. netmon will only check the nodes associated with these IP address periodically, as configured via the Configuration Polling Interval in the Network Polling dialog, to minimize the traffic to these “uninteresting” nodes. If the discovery filter is changed, netmon will clear this “don’t discover” list and immediately schedule polling activities to determine if the nodes on the list pass the filter.

**NOTE:**

Since netmon frequently looks up interfaces based on their IP addresses, it maintains two hash list to speed up access. Each list has 1024 buckets and the hashing function is a simple modulo of the IP address. The hash buckets themselves have a pointer to the in-memory interface structures. The first hash list is for the IP addresses of interfaces that pass the filter and the second is for the “don’t discover” list.

The “don’t discover” list can be accessed by using one of the netmon command-line options to dump its lists. See the Command Line Options section below for details.

**Subnet Layout**

The ability to determine the topological connectivity of bridges and hubs is one of netmon’s major tasks. Although simple in concept, as explained below, the actual implementation is tricky, as each step could be being performed simultaneously for several nodes at a time.
Concept and Approach

The idea is to use the bridge and hub SNMP port tables and their ability to map a source link level address to a physical port to construct a tree of segments with connector devices separating them. This port mapping capability is critical to the design; devices without it cannot be mapped correctly.

For each subnet, a reference or root interface is selected. netmon will choose either an interface of the management station itself, if it has one in the subnet, or the interface of a router.

As part of every configuration operation performed on bridges and hub (called device in the descriptions below), netmon goes through three separate phases.

- **Phase 1 - On which port is the root heard?**
  netmon determines on which physical port device “hears” packets sent from root. This is referred to as the rootport. The interface attaching device to the segment pointing towards root is called device’s highiface; that segment is the device’s highseg.

- **Phase 2 - Is it further away from root than other bridges of hubs?**
  For every other bridge or hub (called target) in device’s highseg, netmon asks target through which port it hears device. If target hears device through target’s rootport, we conclude that device is not further away from target in the tree.
  
  If target hears device on any other port though, netmon conclude device is further away from target in the tree, and device is moved to the segment associated with the port on target, possibly causing the creation of a new segment and restarting this phase with the new segment as highseg.
  
  When netmon concludes that there are no more targets in highseg, the phase ends.

- **Phase 3 - Is it closer to root than other bridges or hubs?**
  For each target bridge or hub in device’s highseg, netmon asks device through which port it hears target.
  
  If device hears target through any port other than rootport, target is moved to the segment associated with the port of device as in the previous phase.
  
  When a non-connector node is added to a network containing HP bridges and hubs, a process quite similar to the phase 2 above is performed on behalf of the node. The process is performed periodically on all nodes in the network, even if no new bridges or hubs have been added. This compensates for topology changes, bridge table overflows and the fact that the port mapping operation does not always succeed.
The algorithm is designed to be redundant. Going through the phase 2 for one device with respect to a second device is identical to performing the phase 3 on that second device with respect to the first.

As part of the configuration operations, netmon also verifies that non-connector nodes are still heard through the same port on the “parent” connector device as before. netmon also verifies that bridges and hubs still hear root through the same rootport. In the port in either of these cases is different, the node is moved to the default segment in the network and phase 2 is restarted.
Command Line Options

Specified via $OV_LRF/netmon.lrf

- **b** count  A burst of pings is sent on last retry. Useful with unreliable networks.

- **J**  Jump start discovery process with broadcast pings from HP agents.

- **r**  Always pull routing tables.

- **R**  Never pull routing tables.
  Default is only for nodes with WAN links.

- **p** timeout  Specify timeout for connector port table entries.
  Default timeout is 15 minutes

- **n** node_name  dumps nodes and associated interfaces

- **i** IP_address  dumps interfaces associated with IP address

- **a** action_nr  special netmon action to dump. Action numbers are:
  
  ?  —valid options
  
  6  —interface hash lists
  
  11—pingWait list
  
  12—pingList
  
  15—snmpWait list
  
  16—snmpList