           Techniques for Managing Asynchronously Generated Alerts

Status of this Memo

   This memo defines common mechanisms for managing asynchronously
   produced alerts in a manner consistent with current network
   management protocols.

   This memo specifies an Experimental Protocol for the Internet
   community. Discussion and suggestions for improvement are requested.
   Please refer to the current edition of the "IAB Official Protocol
   Standards" for the standardization state and status of this protocol.
   Distribution of this memo is unlimited.

Abstract

   This RFC explores mechanisms to prevent a remotely managed entity
   from burdening a manager or network with an unexpected amount of
   network management information, and to ensure delivery of "important"
   information. The focus is on controlling the flow of asynchronously
   generated information, and not how the information is generated.

Table of Contents

1. Introduction

    This memo defines mechanisms to prevent a remotely managed entity
    from burdening a manager or network with an unexpected amount of
    network management information, and to ensure delivery of "important"
    information. The focus is on controlling the flow of asynchronously
    generated information, and not how the information is generated.
    Mechanisms for generating and controlling the generation of
    asynchronous information may involve protocol specific issues.

    There are two understood mechanisms for transferring network
    management information from a managed entity to a manager: request-
    response driven polling, and the unsolicited sending of "alerts".
    Alerts are defined as any management information delivered to a
    manager that is not the result of a specific query. Advantages and
    disadvantages exist within each method. They are detailed in section
    2 below.

    Alerts in a failing system can be generated so rapidly that they
    adversely impact functioning resources. They may also fail to be
    delivered, and critical information maybe lost. Methods are needed
    both to limit the volume of alert transmission and to assist in
    delivering a minimum amount of information to a manager.

It is our belief that managed agents capable of asynchronously
generating alerts should attempt to adopt mechanisms that fill both
of these needs. For reasons shown in section 2.4, it is necessary to
fulfill both alert-management requirements. A complete alert-driven
system must ensure that alerts are delivered or their loss detected
with a means to recreate the lost information, AND it must not allow
itself to overburden its manager with an unreasonable amount of
information.

2. Problem Definition

The following discusses the relative advantages and disadvantages of
polled vs. alert driven management.

2.1 Polling Advantages

(a) Reliable detection of failures.

    A manager that polls for all of its information can
    more readily determine machine and network failures;
    a lack of a response to a query indicates problems
    with the machine or network.   A manager relying on
    notification of problems might assume that a faulty
    system is good, should the alert be unable to reach
    its destination, or the managed system be unable to
    correctly generate the alert. Examples of this
    include network failures (in which an isolated network
    cannot deliver the alert), and power failures (in which
    a failing machine cannot generate an alert). More
    subtle forms of failure in the managed entity might
    produce an incorrectly generated alert, or no alert at
    all.

(b) Reduced protocol complexity on managed entity

    The use of a request-response based system is based on
    conservative assumptions about the underlying transport
    protocol. Timeouts and retransmits (re-requests) can
    be built into the manager. In addition, this allows
    the manager to affect the amount of network management
    information flowing across the network directly.

(c) Reduced performance impact on managed entity

    In a purely polled system, there is no danger of having
    to often test for an alert condition. This testing
    takes CPU cycles away from the real mission of the
    managed entity. Clearly, testing a threshold on each

packet received could have unwanted performance effects
on machines such as gateways. Those who wish to use
thresholds and alerts must choose the parameters to be
tested with great care, and should be strongly
discouraged from updating statistics and checking values
frequently.

   (d) Reduced Configuration Requirements to manage remote
        entity

        Remote, managed entities need not be configured
        with one or more destinations for reporting information.
        Instead, the entity merely responds to whomever
        makes a specific request. When changing the network
        configuration, there is never a need to reconfigure
        all remote manageable systems. In addition, any number
        of "authorized" managers (i.e., those passing any
        authentication tests imposed by the network management
        protocol) may obtain information from any managed entity.
        This occurs without reconfiguring the entity and
        without reaching an entity-imposed limit on the maximum
        number of potential managers.

2.2 Polling Disadvantages

   (a) Response time for problem detection

        Having to poll many MIB [2] variables per machine on
        a large number of machines is itself a real
        problem. The ability of a manager to monitor
        such a system is limited; should a system fail
        shortly after being polled there may be a significant
        delay before it is polled again. During this time,
        the manager must assume that a failing system is
        acceptable. See Appendix A for a hypothetical
        example of such a system.

        It is worthwhile to note that while improving the mean
        time to detect failures might not greatly improve the
        time to correct the failure, the problem will generally
        not be repaired until it is detected. In addition,
        most network managers would prefer to at least detect
        faults before network users start phoning in.

   (b) Volume of network management traffic

        Polling many objects (MIB variables) on many machines
        greatly increases the amount of network management

traffic flowing across the network (see Appendix A).
While it is possible to minimize this through the use
of hierarchies (polling a machine for a general status
of all the machines it polls), this aggravates the
response time problem previously discussed.

2.3 Alert Advantages

(a) Real-time Knowledge of Problems

Allowing the manager to be notified of problems
eliminates the delay imposed by polling many objects/
systems in a loop.

(b) Minimal amount of Network Management Traffic

Alerts are transmitted only due to detected errors.
By removing the need to transfer large amounts of status
information that merely demonstrate a healthy system,
network and system (machine processor) resources may be
freed to accomplish their primary mission.

2.4 Alert Disadvantages

(a) Potential Loss of Critical Information

Alerts are most likely not to be delivered when the
managed entity fails (power supply fails) or the
network experiences problems (saturated or isolated).
It is important to remember that failing machines and
networks cannot be trusted to inform a manager that
they are failing.

(b) Potential to Over-inform the Manager

An "open loop" system in which the flow of alerts to
a manager is fully asynchronous can result in an excess
of alerts being delivered (e.g., link up/down messages
when lines vacillate). This information places an extra
burden on a strained network, and could prevent the
manager from disabling the mechanism generating the
alerts; all available network bandwidth into the manager
could be saturated with incoming alerts.

Most major network management systems strive to use an optimal
combination of alerts and polling. Doing so preserves the advantages
of each while eliminating the disadvantages of pure polling.

3. Specific Goals of this Memo

   This memo suggests mechanisms to minimize the disadvantages of alert
   usage. An optimal system recognizes the potential problems
   associated with sending too many alerts in which a manager becomes
   ineffective at managing, and not adequately using alerts (especially
   given the volumes of data that must be actively monitored with poor
   scaling). It is the author's belief that this is best done by
   allowing alert mechanisms that "close down" automatically when over-
   delivering asynchronous (unexpected) alerts, and that also allow a
   flow of synchronous alert information through a polled log. The use
   of "feedback" (with a sliding window "pin") discussed in section 5
   addresses the former need, while the discussion in section 6 on
   "polled, logged alerts" does the latter.

   This memo does not attempt to define mechanisms for controlling the
   asynchronous generation of alerts, as such matters deal with
   specifics of the management protocol. In addition, no attempt is
   made to define what the content of an alert should be. The feedback
   mechanism does require the addition of a single alert type, but this
   is not meant to impact or influence the techniques for generating any
   other alert (and can itself be generated from a MIB object or the
   management protocol). To make any effective use of the alert
   mechanisms described in this memo, implementation of several MIB
   objects is required in the relevant managed systems. The location of
   these objects in the MIB is under an experimental subtree delegated
   to the Alert-Man working group of the Internet Engineering Task Force
   (IETF) and published in the "Assigned Numbers" RFC [5]. Currently,
   this subtree is defined as

        alertMan ::= { experimental 24 }.

4. Compatibility With Existing Network Management Protocols

   It is the intent of this document to suggest mechanisms that violate
   neither the letter nor the spirit of the protocols expressed in CMOT
   [3] and SNMP [4]. To achieve this goal, each mechanism described
   will give an example of its conformant use with both SNMP and CMOT.

5. Closed Loop "Feedback" Alert Reporting with a "Pin" Sliding
     Window Limit

   One technique for preventing an excess of alerts from being delivered
   involves required feedback to the managed agent. The name "feedback"
   describes a required positive response from a potentially "over-
   reported" manager, before a remote agent may continue transmitting
   alerts at a high rate. A sliding window "pin" threshold (so named
   for the metal on the end of a meter) is established as a part of a

user-defined SNMP trap, or as a managed CMOT event. This threshold
defines the maximum allowable number of alerts ("maxAlertsPerTime")
that may be transmitted by the agent, and the "windowTime" in seconds
that alerts are tested against. Note that "maxAlertsPerTime"
represents the sum total of all alerts generated by the agent, and is
not duplicated for each type of alert that an agent might generate.
Both "maxAlertsPerTime" and "windowTime" are required MIB objects of
SMI [1] type INTEGER, must be readable, and may be writable should
the implementation permit it.

Two other items are required for the feedback technique. The first
is a Boolean MIB object (SMI type is INTEGER, but it is treated as a
Boolean whose only value is zero, i.e., "FALSE") named
"alertsEnabled", which must have read and write access. The second
is a user defined alert named "alertsDisabled". Please see Appendix
B for their complete definitions.

5.1 Use of Feedback

When an excess of alerts is being generated, as determined by the
total number of alerts exceeding "maxAlertsPerTime" within
"windowTime" seconds, the agent sets the Boolean value of
"alertsEnabled" to "FALSE" and sends a single alert of type
"alertsDisabled".

Again, the pin mechanism operates on the sum total of all alerts
generated by the remote system. Feedback is implemented once per
agent and not separately for each type of alert in each agent. While
it is also possible to implement the Feedback/Pin technique on a per
alert-type basis, such a discussion belongs in a document dealing
with controlling the generation of individual alerts.

The typical use of feedback is detailed in the following steps:

    (a) Upon initialization of the agent, the value of
         "alertsEnabled" is set to "TRUE".

    (b) Each time an alert is generated, the value of
         "alertsEnabled" is tested. Should the value be "FALSE",
         no alert is sent. If the value is "TRUE", the alert is
         sent and the current time is stored locally.

    (c) If at least "maxAlertsPerTime" have been generated, the
         agent calculates the difference of time stored for the
         new alert from the time associated with alert generated
         "maxAlertsPerTime" previously. Should this amount be
         less than "windowTime", a single alert of the type
         "alertsDisabled" is sent to the manager and the value of

"alertsEnabled" is then set to "FALSE".

(d) When a manager receives an alert of the type "Alerts-
     Disabled", it is expected to set "alertsEnabled" back
     to "TRUE" to continue to receive alert reports.

5.1.1 Example

In a sample system, the maximum number of alerts any single managed
entity may send the manager is 10 in any 3 second interval. A
circular buffer with a maximum depth of 10 time of day elements is
defined to accommodate statistics keeping.

After the first 10 alerts have been sent, the managed entity tests
the time difference between its oldest and newest alerts. By testing
the time for a fixed number of alerts, the system will never disable
itself merely because a few alerts were transmitted back to back.

The mechanism will disable reporting only after at least 10 alerts
have been sent, and the only if the last 10 all occurred within a 3
second interval. As alerts are sent over time, the list maintains
data on the last 10 alerts only.

5.2 Notes on Feedback/Pin Usage

A manager may periodically poll "alertsEnabled" in case an
"alertsDisabled" alert is not delivered by the network. Some
implementers may also choose to add COUNTER MIB objects to show the
total number of alerts transmitted and dropped by "alertsEnabled"
being FALSE. While these may yield some indication of the number of
lost alerts, the use of "Polled, Logged Alerts" offers a superset of
this function.

Testing the alert frequency need not begin until a minimum number of
alerts have been sent (the circular buffer is full). Even then, the
actual test is the elapsed time to get a fixed number of alerts and
not the number of alerts in a given time period. This eliminates the
need for complex averaging schemes (keeping current alerts per second
as a frequency and redetermining the current value based on the
previous value and the time of a new alert). Also eliminated is the
problem of two back to back alerts; they may indeed appear to be a
large number of alerts per second, but the fact remains that there
are only two alerts. This situation is unlikely to cause a problem
for any manager, and should not trigger the mechanism.

Since alerts are supposed to be generated infrequently, maintaining
the pin and testing the threshold should not impact normal
performance of the agent (managed entity). While repeated testing

may affect performance when an excess of alerts are being
transmitted, this effect would be minor compared to the cost of
generating and sending so many alerts. Long before the cost of
testing (in CPU cycles) becomes relatively high, the feedback
mechanism should disable alert sending and affect savings both in
alert sending and its own testing (note that the list maintenance and
testing mechanisms disable themselves when they disable alert
reporting). In addition, testing the value of "alertsEnabled" can
limit the CPU burden of building alerts that do not need to be sent.

It is advised that the implementer consider allowing write access to
both the window size and the number of alerts allowed in a window's
time. In doing so, a management station has the option of varying
these parameters remotely before setting "alertsEnabled" to "TRUE".
Should either of these objects be set to 0, a conformant system will
disable the pin and feedback mechanisms and allow the agent to send
all of the alerts it generates.

While the feedback mechanism is not high in CPU utilization costs,
those implementing alerts of any kind are again cautioned to exercise
care that the alerts tested do not occur so frequently as to impact
the performance of the agent's primary function.

The user may prefer to send alerts via TCP to help ensure delivery of
the "alerts disabled" message, if available.

The feedback technique is effective for preventing the over-reporting
of alerts to a manager. It does not assist with the problem of
"under-reporting" (see "polled, logged alerts" for this).

It is possible to lose alerts while "alertsEnabled" is "FALSE".
Ideally, the threshold of "maxAlertsPerTime" should be set
sufficiently high that "alertsEnabled" is only set to "FALSE" during
"over-reporting" situations. To help prevent alerts from possibly
being lost when the threshold is exceeded, this method can be
combined with "polled, logged alerts" (see below).

6. Polled, Logged Alerts

A simple system that combines the request-response advantages of
polling while minimizing the disadvantages is "Polled, Logged
Alerts". Through the addition of several MIB objects, one gains a
system that minimizes network management traffic, lends itself to
scaling, eliminates the reliance on delivery, and imposes no
potential over-reporting problems inherent in pure alert driven
architectures. Minimizing network management traffic is affected by
reducing multiple requests to a single request. This technique does
not eliminate the need for polling, but reduces the amount of data

transferred and ensures the manager either alert delivery or
notification of an unreachable node. Note again, the goal is to
address the needs of information (alert) flow and not to control the
local generation of alerts.

6.1 Use of Polled, Logged Alerts

As alerts are generated by a remote managed entity, they are logged
locally in a table. The manager may then poll a single MIB object to
determine if any number of alerts have been generated. Each poll
request returns a copy of an "unacknowledged" alert from the alert
log, or an indication that the table is empty. Upon receipt, the
manager might "acknowledge" any alert to remove it from the log.
Entries in the table must be readable, and can optionally allow the
user to remove them by writing to or deleting them.

This technique requires several additional MIB objects. The
alert_log is a SEQUENCE OF logTable entries that must be readable,
and can optionally have a mechanism to remove entries (e.g., SNMP set
or CMOT delete). An optional read-only MIB object of type INTEGER,
"maxLogTableEntries" gives the maximum number of log entries the
system will support. Please see Appendix B for their complete
definitions.

The typical use of Polled, Logged Alerts is detailed below.

   (a) Upon initialization, the agent builds a pointer to a log
        table. The table is empty (a sequence of zero entries).

   (b) Each time a local alert is generated, a logTable entry
        is built with the following information:

   SEQUENCE {
           alertId   INTEGER,
           alertData   OPAQUE
        }

        (1) alertId number of type INTEGER, set to 1 greater
            than the previously generated alertId. If this is
            the first alert generated, the value is initialized
            to 1. This value should wrap (reset) to 1 when it
            reaches 2^{32}. Note that the maximum log depth
            cannot exceed $(2^{32})-1$ entries.

        (2) a copy of the alert encapsulated in an OPAQUE.

   (c) The new log element is added to the table. Should
        addition of the element exceed the defined maximum log

table size, the oldest element in the table (having the lowest alertId) is replaced by the new element.

(d) A manager may poll the managed agent for either the next alert in the alert_table, or for a copy of the alert associated with a specific alertId. A poll request must indicate a specific alertId. The mechanism for obtaining this information from a table is protocol specific, and might use an SNMP GET or GET NEXT (with GET NEXT following an instance of zero returning the first table entry's alert) or CMOT's GET with scoping and filtering to get alertData entries associated with alertId's greater or less than a given instance.

(e) An alertData GET request from a manager must always be responded to with a reply of the entire OPAQUE alert (SNMP TRAP, CMOT EVENT, etc.) or a protocol specific reply indicating that the get request failed.

Note that the actual contents of the alert string, and the format of those contents, are protocol specific.

(f) Once an alert is logged in the local log, it is up to the individual architecture and implementation whether or not to also send a copy asynchronously to the manager. Doing so could be used to redirect the focus of the polling (rather than waiting an average of 1/2 the poll cycle to learn of a problem), but does not result in significant problems should the alert fail to be delivered.

(g) Should a manager request an alert with alertId of 0, the reply shall be the appropriate protocol specific error response.

(h) If a manager requests the alert immediately following the alert with alertId equal to 0, the reply will be the first alert (or alerts, depending on the protocol used) in the alert log.

(i) A manager may remove a specific alert from the alert log by naming the alertId of that alert and issuing a protocol specific command (SET or DELETE). If no such alert exists, the operation is said to have failed and such failure is reported to the manager in a protocol specific manner.

6.1.1 Example

   In a sample system (based on the example in Appendix A), a manager
   must monitor 40 remote agents, each having between 2 and 15
   parameters which indicate the relative health of the agent and the
   network. During normal monitoring, the manager is concerned only
   with fault detection. With an average poll request-response time of
   5 seconds, the manager polls one MIB variable on each node. This
   involves one request and one reply packet of the format specified in
   the XYZ network management protocol. Each packet requires 120 bytes
   "on the wire" (requesting a single object, ASN.1 encoded, IP and UDP
   enveloped, and placed in an ethernet packet). This results in a
   serial poll cycle time of 3.3 minutes (40 nodes at 5 seconds each is
   200 seconds), and a mean time to detect alert of slightly over 1.5
   minutes. The total amount of data transferred during a 3.3 minute
   poll cycle is 9600 bytes (120 requests and 120 replies for each of 40
   nodes). With such a small amount of network management traffic per
   minute, the poll rate might reasonably be doubled (assuming the
   network performance permits it). The result is 19200 bytes
   transferred per cycle, and a mean time to detect failure of under 1
   minute. Parallel polling obviously yields similar improvements.

   Should an alert be returned by a remote agent's log, the manager
   notifies the operator and removes the element from the alert log by
   setting it with SNMP or deleting it with CMOT. Normal alert
   detection procedures are then followed. Those SNMP implementers who
   prefer to not use SNMP SET for table entry deletes may always define
   their log as "read only". The fact that the manager made a single
   query (to the log) and was able to determine which, if any, objects
   merited special attention essentially means that the status of all
   alert capable objects was monitored with a single request.

   Continuing the above example, should a remote entity fail to respond
   to two successive poll attempts, the operator is notified that the
   agent is not reachable. The operator may then choose (if so
   equipped) to contact the agent through an alternate path (such as
   serial line IP over a dial up modem). Upon establishing such a
   connection, the manager may then retrieve the contents of the alert
   log for a chronological map of the failure's alerts. Alerts
   undelivered because of conditions that may no longer be present are
   still available for analysis.

6.2 Notes on Polled, Logged Alerts

   Polled, logged alert techniques allow the tracking of many alerts
   while actually monitoring only a single MIB object. This
   dramatically decreases the amount of network management data that
   must flow across the network to determine the status. By reducing

the number of requests needed to track multiple objects (to one), the
poll cycle time is greatly improved. This allows a faster poll cycle
(mean time to detect alert) with less overhead than would be caused
by pure polling.

In addition, this technique scales well to large networks, as the
concept of polling a single object to learn the status of many lends
itself well to hierarchies. A proxy manager may be polled to learn
if he has found any alerts in the logs of the agents he polls. Of
course, this scaling does not save on the mean time to learn of an
alert (the cycle times of the manager and the proxy manager must be
considered), but the amount of network management polling traffic is
concentrated at lower levels. Only a small amount of such traffic
need be passed over the network's "backbone"; that is the traffic
generated by the request-response from the manager to the proxy
managers.

Note that it is best to return the oldest logged alert as the first
table entry. This is the object most likely to be overwritten, and
every attempt should be made ensure that the manager has seen it. In
a system where log entries may be removed by the manager, the manager
will probably wish to attempt to keep all remote alert logs empty to
reduce the number of alerts dropped or overwritten. In any case, the
order in which table entries are returned is a function of the table
mechanism, and is implementation and/or protocol specific.

"Polled, logged alerts" offers all of the advantages inherent in
polling (reliable detection of failures, reduced agent complexity
with UDP, etc.), while minimizing the typical polling problems
(potentially shorter poll cycle time and reduced network management
traffic).

Finally, alerts are not lost when an agent is isolated from its
manager. When a connection is reestablished, a history of conditions
that may no longer be in effect is available to the manager. While
not a part of this document, it is worthwhile to note that this same
log architecture can be employed to archive alert and other
information on remote hosts. However, such non-local storage is not
sufficient to meet the reliability requirements of "polled, logged
alerts".

7. Compatibility with SNMP [4] and CMOT [3]

7.1 Closed Loop (Feedback) Alert Reporting

7.1.1 Use of Feedback with SNMP

   At configuration time, an SNMP agent supporting Feedback/Pin is
   loaded with default values of "windowTime" and "maxAlerts-PerTime",
   and "alertsEnabled" is set to TRUE. The manager issues an SNMP GET
   to determine "maxAlertsPerTime" and "windowTime", and to verify the
   state of "alertsEnabled". Should the agent support setting Pin
   objects, the manager may choose to alter these values (via an SNMP
   SET). The new values are calculated based upon known network
   resource limitations (e.g., the amount of packets the manager's
   gateway can support) and the number of agents potentially reporting
   to this manager.

   Upon receipt of an "alertsDisabled" trap, a manager whose state and
   network are not overutilized immediately issues an SNMP SET to make
   "alertsEnabled" TRUE. Should an excessive number of "alertsDisabled"
   traps regularly occur, the manager might revisit the values chosen
   for implementing the Pin mechanism. Note that an overutilized system
   expects its manager to delay the resetting of "alertsEnabled".

   As a part of each regular polling cycle, the manager includes a GET
   REQUEST for the value of "alertsEnabled". If this value is FALSE, it
   is SET to TRUE, and the potential loss of traps (while it was FALSE)
   is noted.

7.1.2 Use of Feedback with CMOT

   The use of CMOT in implementing Feedback/Pin is essentially identical
   to the use of SNMP. CMOT GET, SET, and EVENT replace their SNMP
   counterparts.

7.2 Polled, Logged Alerts

7.2.1 Use of Polled, Logged alerts with SNMP

   As a part of regular polling, an SNMP manager using Polled, logged
   alerts may issue a GET_NEXT Request naming
   { alertLog logTableEntry(1) alertId(1) 0 }. Returned is either the
   alertId of the first table entry or, if the table is empty, an SNMP
   reply whose object is the "lexicographical successor" to the alert
   log.

   Should an "alertId" be returned, the manager issues an SNMP GET
   naming { alertLog logTableEntry(1) alertData(2) value } where "value"

is the alertId integer obtained from the previously described GET
NEXT. This returns the SNMP TRAP encapsulated within an OPAQUE.

If the agent supports the deletion of table entries through SNMP
SETS, the manager may then issue a SET of { alertLog logTableEntry(1)
alertId(1) value } to remove the entry from the log. Otherwise, the
next GET NEXT poll of this agent should request the first "alertId"
following the instance of "value" rather than an instance of "0".

7.2.2 Use of Polled, Logged Alerts with CMOT

Using polled, logged alerts with CMOT is similar to using them with
SNMP. In order to test for table entries, one uses a CMOT GET and
specifies scoping to the alertLog. The request is for all table
entries that have an alertId value greater than the last known
alertId, or greater than zero if the table is normally kept empty by
the manager. Should the agent support it, entries are removed with a
CMOT DELETE, an object of alertLog.entry, and a distinguishing
attribute of the alertId to remove.

8. Multiple Manager Environments

The conflicts between multiple managers with overlapping
administrative domains (generally found in larger networks) tend to
be resolved in protocol specific manners. This document has not
addressed them. However, real world demands require alert management
techniques to function in such environments.

Complex agents can clearly respond to different managers (or managers
in different "communities") with different reply values. This allows
feedback and polled, logged alerts to appear completely independent
to differing autonomous regions (each region sees its own value).
Differing feedback thresholds might exist, and feedback can be
actively blocking alerts to one manager even after another manager
has reenabled its own alert reporting. All of this is transparent to
an SNMP user if based on communities, or each manager can work with a
different copy of the relevant MIB objects. Those implementing CMOT
might view these as multiple instances of the same feedback objects
(and allow one manager to query the state of another's feedback
mechanism).

The same holds true for polled, logged alerts. One manager (or
manager in a single community/region) can delete an alert from its
view without affecting the view of another region's managers.

Those preferring less complex agents will recognize the opportunity
to instrument proxy management. Alerts might be distributed from a
manager based alert exploder which effectively implements feedback

and polled, logged alerts for its subscribers. Feedback parameters
are set on each agent to the highest rate of any subscriber, and
limited by the distributor. Logged alerts are deleted from the view
at the proxy manager, and truly deleted at the agent only when all
subscribers have so requested, or immediately deleted at the agent
with the first proxy request, and maintained as virtual entries by
the proxy manager for the benefit of other subscribers.

9. Summary

   While "polled, logged alerts" may be useful, they still have a
   limitation: the mean time to detect failures and alerts increases
   linearly as networks grow in size (hierarchies offer shorten
   individual poll cycle times, but the mean detection time is the sum
   of 1/2 of each cycle time). For this reason, it may be necessary to
   supplement asynchronous generation of alerts (and "polled, logged
   alerts") with unrequested transmission of the alerts on very large
   networks.

   Whenever systems generate and asynchronously transmit alerts, the
   potential to overburden (over-inform) a management station exists.
   Mechanisms to protect a manager, such as the "Feedback/Pin"
   technique, risk losing potentially important information. Failure to
   implement asynchronous alerts increases the time for the manager to
   detect and react to a problem. Over-reporting may appear less
   critical (and likely) a problem than under-informing, but the
   potential for harm exists with unbounded alert generation.

   An ideal management system will generate alerts to notify its
   management station (or stations) of error conditions. However, these
   alerts must be self limiting with required positive feedback. In
   addition, the manager should periodically poll to ensure connectivity
   to remote stations, and to retrieve copies of any alerts that were
   not delivered by the network.

10. References

   [1] Rose, M., and K. McCloghrie, "Structure and Identification of
       Management Information for TCP/IP-based Internets", RFC 1155,
       Performance Systems International and Hughes LAN Systems, May
       1990.

   [2] McCloghrie, K., and M. Rose, "Management Information Base for
       Network Management of TCP/IP-based internets", RFC 1213, Hughes
       LAN Systems, Inc., Performance Systems International, March 1991.

   [3] Warrier, U., Besaw, L., LaBarre, L., and B. Handspicker, "Common
       Management Information Services and Protocols for the Internet

     (CMOT) and (CMIP)", RFC 1189, Netlabs, Hewlett-Packard, The Mitre
     Corporation, Digital Equipment Corporation, October 1990.

  [4] Case, J., Fedor, M., Schoffstall, M., and C. Davin, "Simple
     Network Management Protocol" RFC 1157, SNMP Research, Performance
     Systems International, Performance Systems International, MIT
     Laboratory for Computer Science, May 1990.

  [5] Reynolds, J., and J. Postel, "Assigned Numbers", RFC 1060,
     USC/Information Sciences Institute, March 1990.

11. Acknowledgements

Appendix A

  Example of polling costs

     The following example is completely hypothetical, and arbitrary.
     It assumes that a network manager has made decisions as to which
     systems, and which objects on each system, must be continuously
     monitored to determine the operational state of a network. It
     does not attempt to discuss how such decisions are made, and
     assumes that they were arrived at with the full understanding that
     the costs of polling many objects must be weighed against the

level of information required.

Consider a manager that must monitor 40 gateways and hosts on a
single network. Further assume that the average managed entity
has 10 MIB objects that must be watched to determine the device's
and network's overall "health". Under the XYZ network management
protocol, the manager may get the values of up to 4 MIB objects
with a single request (so that 3 requests must be made to
determine the status of a single entity). An average response
time of 5 seconds is assumed, and a lack of response within 30
seconds is considered no reply. Two such "no replies" are needed
to declare the managed entity "unreachable", as a single packet
may occasionally be dropped in a UDP system (those preferring to
use TCP for automated retransmits should assume a longer timeout
value before declaring the entity "unreachable" which we will
define as 60 seconds).

We begin with the case of "sequential polling". This is defined
as awaiting a response to an outstanding request before issuing
any further requests. In this example, the average XYZ network
management protocol packet size is 300 bytes "on the wire"
(requesting multiple objects, ASN.1 encoded, IP and UDP enveloped,
and placed in an ethernet packet). 120 request packets are sent
each cycle (3 for each of 40 nodes), and 120 response packets are
expected. 72000 bytes (240 packets at 300 bytes each) must be
transferred during each poll cycle, merely to determine that the
network is fine.

At five seconds per transaction, it could take up to 10 minutes to
determine the state of a failing machine (40 systems x 3 requests
each x 5 seconds per request). The mean time to detect a system
with errors is 1/2 of the poll cycle time, or 5 minutes. In a
failing network, dropped packets (that must be timed out and
resent) greatly increase the mean and worst case times to detect
problems.

Note that the traffic costs could be substantially reduced by
combining each set of three request/response packets in a single
request/response transaction (see section 6.1.1 "Example").

While the bandwidth use is spread over 10 minutes (giving a usage
of 120 bytes/second), this rapidly deteriorates should the manager
decrease his poll cycle time to accommodate more machines or
improve his mean time to fault detection. Conversely, increasing
his delay between polls reduces traffic flow, but does so at the
expense of time to detect problems.

Many network managers allow multiple poll requests to be "pending"

at any given time. It is assumed that such managers would not
normally poll every machine without any delays. Allowing
"parallel polling" and initiating a new request immediately
following any response would tend to generate larger amounts of
traffic; "parallel polling" here produces 40 times the amount of
network traffic generated in the simplistic case of "sequential
polling" (40 packets are sent and 40 replies received every 5
seconds, giving 80 packets x 300 bytes each per 5 seconds, or 4800
bytes/second). Mean time to detect errors drops, but at the cost
of increased bandwidth. This does not improve the timeout value
of over 2 minutes to detect that a node is not responding.

Even with parallel polling, increasing the device count (systems
to manage) not only results in more traffic, but can degrade
performance. On large networks the manager becomes bounded by the
number of queries that can be built, tracked, responses parsed,
and reacted to per second. The continuous volume requires the
timeout value to be increased to accommodate responses that are
still in transit or have been received and are queued awaiting
processing. The only alternative is to reduce the poll cycle.
Either of these actions increase both mean time to detect failure
and worst case time to detect problems.

If alerts are sent in place of polling, mean time to fault
detection drops from over a minute to as little as 2.5 seconds
(1/2 the time for a single request-response transaction). This
time may be increased slightly, depending on the nature of the
problem. Typical network utilization is zero (assuming a
"typical" case of a non-failing system).

Appendix B

All defined MIB objects used in this document reside
under the mib subtree:

alertMan ::= { iso(1) org(3) dod(6) internet(1)
     experimental(3) alertMan(24) ver1(1) }

as defined in the Internet SMI [1] and the latest "Assigned
Numbers" RFC [5]. Objects under this branch are assigned
as follows:

RFC 1224-MIB DEFINITIONS ::= BEGIN

alertMan   OBJECT IDENTIFIER ::= { experimental 24 }

ver1   OBJECT IDENTIFIER ::= { alertMan 1 }

```
feedback          OBJECT IDENTIFIER ::= { ver1 1 }
polledLogged      OBJECT IDENTIFIER ::= { ver1 2 }

END
```

1) Feedback Objects

   OBJECT:
   ------

   maxAlertsPerTime { feedback 1 }

      Syntax:
         Integer

      Access:
         read-write

      Status:
         mandatory

   OBJECT:
   ------

   windowTime { feedback 2 }

      Syntax:
         Integer

      Access:
         read-write

      Status:
         mandatory

   OBJECT:
   ------

   alertsEnabled { feedback 3 }

      Syntax:
         Integer

      Access:
         read-write

      Status:

```

          mandatory

     2) Polled, Logged Objects

     OBJECT:
     ------

     alertLog { polledLogged 1 }

     Syntax:
        SEQUENCE OF logTableEntry

     Access:
        read-write

     Status:
        mandatory

     OBJECT:
     ------

     logTableEntry { alertLog 1 }

     Syntax:

        logTableEntry ::= SEQUENCE {

           alertId
              INTEGER,
           alertData
              OPAQUE
        }

     Access:
        read-write

     Status:
        mandatory

     OBJECT:
     ------

     alertId { logTableEntry 1 }

     Syntax:
        Integer

Access:
    read-write

Status:
    mandatory

OBJECT:
------

alertData { logTableEntry 2 }

Syntax:
    Opaque

Access:
    read-only

Status:
    mandatory

OBJECT:
------

maxLogTableEntries { polledLogged 2 }

Syntax:
    Integer

Access:
    read-only

Status:
    optional

Security Considerations

    Security issues are not discussed in this memo.

Author's Address

    Lou Steinberg
    IBM NSFNET Software Development
    472 Wheelers Farms Rd, m/s 91
    Milford, Ct. 06460

    Phone:      203-783-7175
    EMail:      LOUISS@IBM.COM