The Annotated `RMON` Developer's Guide

R. Kooijman & D.M. Wisse

September 1, 1993

Contents

1 Introduction **2**

2 RMON **MIB** **3**

3 Open Issues in RFC 1271 **6**

4 Implementation Guidelines **7**

4.1 Data Types 7

4.2 Return Codes 7

4.3 Default Values for Variables 8

4.4 Configuration Values 8

4.5 Row Creation 8

4.6 `History` 9

4.6.1 `EtherHistoryEntry` 9

4.7 `Alarm` 9

4.8 `Filter` 10

4.9 `Host` 10

4.9.1 Data structures 10

4.10 `TopN` 11

4.10.1 Data structures 12

4.11 `Event` 12

4.12 Filtering through Channels 12

4.13 Interdependencies 13

5 Row Creation Problems **17**

5.1 SNMP Retries 17

5.2 Multiple Manager Issues 17

6 Frequently Asked Questions **20**

7 Conclusions **21**

Acknowledgements

I would like to thank William Stallings for his discussion about our filtering algorithm which resulted in a better, faster and simpler method. I also would like to thank Brian Wyld for informing me about the method for setting `DataSource`'s to `Channel`'s to get filtered data in a 'standardized' manner. Last but not least, I want to thank Lyle Weiman for suggesting several changes to this document, of which the addition of discussions on multiple manager instance and row creation problems were the most important.

Chapter 1

Introduction

The RMON MIB was defined by the Internet Network Working Group to make information of remote network monitors available for manager applications in a standardized manner.

This report tries to be a complement to the RMON MIB itself by describing and clarifying some vague parts of the standard and by providing some guidelines for implementing RMON compliant network monitors. Most of the details are a direct result from our own decisions that were made during the implementation of the Beholder TNG network monitor [1].

First, the RMON MIB itself and its use are described. After that, we will present some guidelines on how to implement an RMON agent and we will make some remarks on the more vague parts of this MIB. Finally, we will discuss some problems one has to be aware of when an RMON agent is used in a real-life network and we will draw some conclusions.

Chapter 2

RMON MIB

The RMON MIB defines the following groups [2]:

statistics The statistics group contains statistics measured by the probe for each monitored interface on this device.

history The history group records periodically a limited number of statistical samples from a network and stores them for later retrieval.

alarm The alarm group periodically takes statistical samples from variables in the probe and compares them to previously configured thresholds. If the monitored variable crosses a threshold, an event is generated. A hysteresis mechanism is implemented to limit the generation of alarms.

hosts The host group contains simple statistics associated with each host discovered on the network. This group discovers hosts on the network by keeping a list of source and destination MAC Addresses seen in good packets promiscuously received from the network.

hostTopN The host Top N group is used to prepare reports that describe the hosts that top a list ordered by one of their statistics. The available statistics are samples of one of their base statistics over an interval specified by the management station. Thus, these statistics are rate based. The management station also selects how many such hosts are reported.

matrix The matrix group stores statistics for conversations between two MAC addresses. As the device detects a new conversation, it creates a new entry in its tables.

filter The filter group allows packets to be matched by a filter equation. These matched packets form a data stream that may be captured or may generate events.

capture The packet capture group allows packets to be captured after they flow through a channel.

event The event group controls the generation and notification of events from this device.

Specific instances can be created and deleted dynamically by managers in a RMON agent. For this purpose, each group has a `Owner` and a `Status` variable. The `Owner` variable is of type `DisplayString` and is used to designate the owner of the instance. The `Status` variable is used to create, start and delete instances.

Table 2.1 shows an overview of values to which the `Status` variable can be set, dependent upon the current value of the variable. Shown are the SNMP return codes and the value that `Status` will have when retrieved again for the first time if that is different from the value to which it was set. This table assumes a perfect connection between manager and agent without any retries.

Table 2.1: `Status` value matrix

Status			
current value	value set to		
	INVALID	CREATE REQUEST	VALID
INVALID	NO SUCH NAME	NO ERROR collector created Status: UNDER CREATION	NO SUCH NAME
UNDER CREATION	NO ERROR collector deleted Status: INVALID	BAD VALUE	NO ERROR collector started Status: VALID
VALID	NO ERROR collector deleted Status: INVALID	BAD VALUE	NO ERROR

SNMP error codes may be returned in case the monitor can't perform a request. The issue of what code to return when and the situation with retries are discussed in more detail in section 4.2 and 5.1.

The list of actions a manager application normally follows to create an instance is:

1. Find an available index number for the instance that is to be created
2. Set the `Status` of the instance with the found index to CREATE REQUEST
3. Fill in all necessary configuration variables (if successful)
4. Start the instance by setting its `Status` to VALID

After the manager is done with the instance it created, it should also delete it again by setting the Status of the instance to INVALID.

This scenario is only applicable when no errors occur or when the agent is used by one manager application only. The problems that can occur in case of retries or when more than one manager application is using the RMON agent are described in more detail in chapter 5.

Finding an available index can be done in different ways, using:

- the sequential approach,

- the MIB-next approach or
- the pseudo-random approach

Using the sequential approach, a manager application starts at index 1 and increases the index each time it finds that an instance with that number already exists. The manager will ultimately find the lowest available index.

A manager that uses the MIB-next approach will perform a get-next operation on the `Status` variables of the group in which it wants to create a new instance until it finds no more entries. The last found index + 1 will then be used as the new index. This way, the manager will create an instance with an index that is higher than any of the others in the same group.

Both these approaches have the disadvantage that looking for a new index can take some time and can create some network load: most of the time all existing instances in a group must be checked through SNMP queries. Using the pseudo-random number approach, however, a manager application tries to find an available instance index by generating pseudo-random numbers for it.

The choice of which approach to use is also important in multiple manager environments where row creation collisions can occur. More details about this subject can be found in section 5.2.

Chapter 3

Open Issues in RFC 1271

Several issues are left open for discussion in RFC 1271:

- Explicit definition of what error codes to return when.
- Specific default values of certain variables.
- The necessity or usefulness of internal checking of successful event generation.
- The accuracy in building a `TopN` list.
- Interdependencies between `RMON` groups.
- Return values of variables without defaults.

Our opinions on these and other issues will be discussed in the next chapter.

Chapter 4

Implementation Guidelines

4.1 Data Types

Table 4.1 shows some SNMP types with the recommended internal data types to be used in a RMON monitor.

Table 4.1: Recommended internal data types

SNMP type	internal data type
INTEGER	32 bits signed integer
OCTET STRING	array of bytes
OBJECT IDENTIFIER	array of 32 bits signed integers
Counter	32 bits unsigned integer
TimeTicks	32 bits unsigned integer
DisplayString	array of characters
return code	16 bits unsigned integer

4.2 Return Codes

RFC 1271 doesn't specify what SNMP error codes should be returned when. It is often difficult to decide what to return in situations like when the agent runs out of memory or when a manager tries to validate/start a collector that is missing some important information.

If the following rules are applied, the RMON monitor will display a consistent behavior:

- If a collector needs to allocate memory and it fails to do so, return a GENERIC error.
- If a bad value is assigned to a collector parameter, return a BAD VALUE error.

- If a collector is started (by setting its status to VALID) and it misses vital information or has parameters set to values that are not applicable, then return a BAD VALUE error.

SNMP v2 is a big improvement by defining error codes that can indicate out-of-memory and other problems.

4.3 Default Values for Variables

The default values for a number of variables isn't defined in RFC 1271 while others are. We think all of them should be defined or none. Without default values all variables in a collector should be set by the manager. Until the moment they are set, they stay invisible (a get request on these variables will return NO SUCH NAME) and the collector may not be started.

4.4 Configuration Values

RFC 1271 speaks of vague situations such as if an agent runs out of resources. Sometimes one wants to restrict certain tables in the monitor on purpose, so those tables don't hog your whole computer. It is therefore wise to make configurable maxima e.g. for the number of log entries, the size of the host table and the size of the matrix table.

4.5 Row Creation

Closely related to default values is the subject of row creation. RFC 1271 doesn't prescribe what to do with rows of collectors that have been created but don't have a default value nor have they been set yet by a manager. What does one return in case they are retrieved by a manager?

There are 2 possibilities in such a case:

- Define a default value and return this until the value is changed.
- Return NO SUCH NAME until the variable is set explicitly to a value by a manager.

We prefer the first approach: satisfying default values can almost always be defined. Most of the time integers can be set to 0, booleans to true(1) or false(2), etc.

4.6 History

4.6.1 EtherHistoryEntry

An Ethernet frame has the global layout as in figure 4.1.

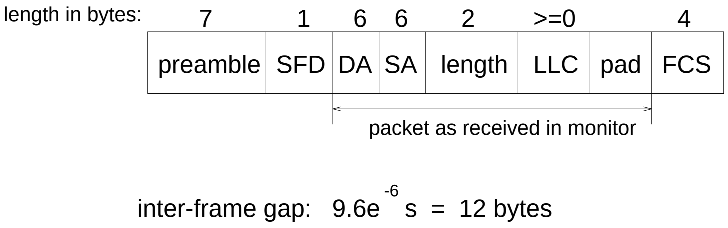

Figure 4.1: Ethernet frame layout

Besides the number of bytes that have come along, one must also take the preamble, SFD, FCS and inter-gap spacing into account when estimating the utilization (in hundredths of a percent) of an Ethernet network segment:

$$Utilization = \frac{(Octets + Pkts \times 24) * 8}{C_{Ethernet} \times T_{sample}} \times 10000 \tag{4.1}$$

where $C_{Ethernet}$ stands for the capacity of an Ethernet type of network and T_{sample} is the actual time (in seconds) between two consecutive samples when the number of octets and packets is determined.

4.7 Alarm

There is an unresolved issue in the `Alarm` group: It is not sure what one should do with `alarmStartupAlarm` in case `alarmSampleType` is set to `deltaValue(2)`. There are two possibilities:

1. Don't use `alarmStartupAlarm` at all in case the `Alarm` entry is set up to do delta sampling.
2. Generate a possible event only if the `Alarm` entry has acquired the necessary samples to do a justified check.

We recommend method 1 because method 2 can take a long time if the alarm monitoring interval is large while it should be generated right after the start of the collector.

4.8 Filter

The filters as described in RFC 1271 are very simple. You can specify that certain data values at certain offsets in received packets need to be matched or not. It would be far more versatile if you could specify certain data values to be matched (or not) in certain fields of a specific protocol. In case of protocols with varying lengths of fields it is even necessary. Therefore, we suggest a MIB-like protocol setup: protocol fields can then be designated by identifiers.

Below is pseudo code for the filtering algorithm as described in RFC 1271.

Algorithm 1 *Filtering*

```
if PktDataLen == 0 then
    no match
end
if PktDataOffset + PktDataOffset > Bytes(packet) then
    no match
end
rule2fail = FALSE
if PktDataNotMaskAllZero == TRUE then
    rule3fail = FALSE
else
    rule3fail = TRUE
end
for bi = first_byte; bi < last_byte; bi++
    scratch = packet[bi]^PktData[bi-first_byte] & PktDataMask[bi-first_byte]
    if (scratch &~data->PktDataNotMask[bi-first_byte]) != 0
        rule2fail = TRUE
    end
    if rule3fail == TRUE && (scratch & PktDataNotMask[bi-first_byte]) != 0
        rule3fail = FALSE
    end
end
if rule2fail == FALSE && rule3fail == FALSE
    packet match
end
```

end

4.9 Host

4.9.1 Data structures

It is best to use a hash table to store `Host` entries. Such a `Host` entry consists of the following data elements:

- host address
- number of packets received by host
- number of packets sent by host
- number of octets received by host
- number of octets sent by host
- number of broadcast packets sent by host
- number of multicast packets sent by host
- number of error packets sent by host

These entries are furthermore ordered using 3 lists:

- a lexically ordered list
- a list ordered on the time of detection
- a least recently used ordered list

Figure 4.2 depicts these datastructures.

4.10 `TopN`

There are two approaches to calculate a `TopN` list: the snapshot method and the per-packet method (figure 4.3 depicts these 2 methods):

- Snapshot method
  In this method two snapshots are taken (one at the beginning of the sampling period and one at the end) from the host table and its relevant contents and subtracted from one another.
- Packet Update method
  In this method the `TopN` list is updated every time host information is updated in the corresponding `Host` table. This method involves more CPU overhead per packet but is at the same time more accurate.

The big difference between these two methods is the measure of accuracy. The snapshot method will fail to take into account the following situations:

- Counters that have wrapped between the first and the last snapshot.
- Hosts that have temporarily disappeared from the `Host` table (counters are reset after reappearance but this will not be noticed).

Small note: The `RMON` MIB defines hostTopNRate as `INTEGER` but in the implementation you should use a data type equivalent to the one used for `Counters` because otherwise the intermediate calculation for determining the change can overflow as the statistics in `Host` entries are of type `Counter`.

4.10.1 Data structures

Like in the `Host` datastructures, a hash table can be used best to store `TopN` entries. A `TopN` entry has the following elements:

- host address
- number of packets received by host
- number of packets sent by host
- number of octets received by host
- number of octets sent by host
- number of broadcast packets sent by host
- number of multicast packets sent by host
- number of error packets sent by host

The numbers are all counted since the start of the `TopN` collector.

Figure 4.4 depicts these datastructures.

4.11 `Event`

Certain groups in the `RMON` MIB may generate events but these may fail if the designated events don't exist. It isn't clear from RFC 1271 whether the successful generation of an event should be checked internally.

Advantages in checking this:

- `channelEventStatus` will only be set to `eventFired` if the generation of the event was actually successful.
- The hysteresis mechanism can take the failure of generations into account: if an event generation fails the mechanism can return to a state where both rising and falling events will be allowed again, and not only the opposite event if it should be successful.

4.12 Filtering through Channels

Some agents have the feature to filter the input of collector groups that built statistics based on packet streams, by means of attaching a channel to their data source instead of the interface itself. For example, in such agents, one can set `EtherStatsDataSource` to 1.3.6.1.2.1.16.7.2.1.1 instead of 1.3.6.1.2.1.2.2.1.1.1 to use the filtered packet stream of `Channel` 1 and not the raw packet stream of the 1st interface.

This is a very useful feature which is not mentioned explicitly in RFC 1271.

4.13 Interdependencies

The only dependency between RMON groups that is defined by RFC 1271 is the dependency between an Alarm instance and the variable it monitors: the variable must exist otherwise the Alarm instance should be automatically removed.

Defining interdependencies contradicts with the basic principle behind SNMP agents: simplicity.

Some argue that dependencies can protect against multiple-manager race conditions. However, this problem can never be completely resolved in the current setup of the RMON MIB, so partly protection doesn't help much either.

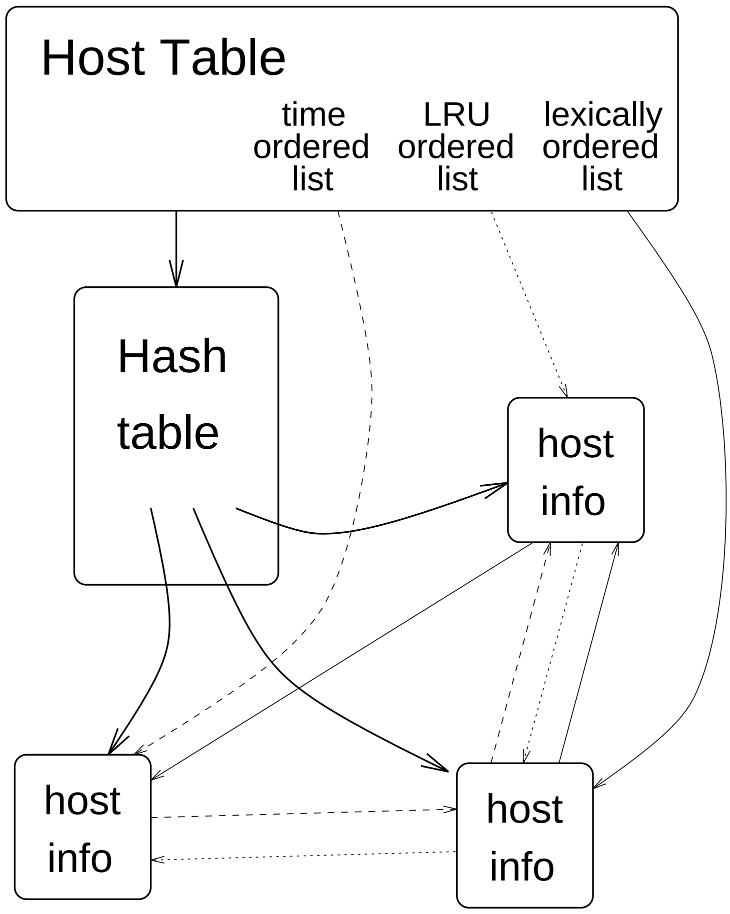

Figure 4.2: `Host` datastructures

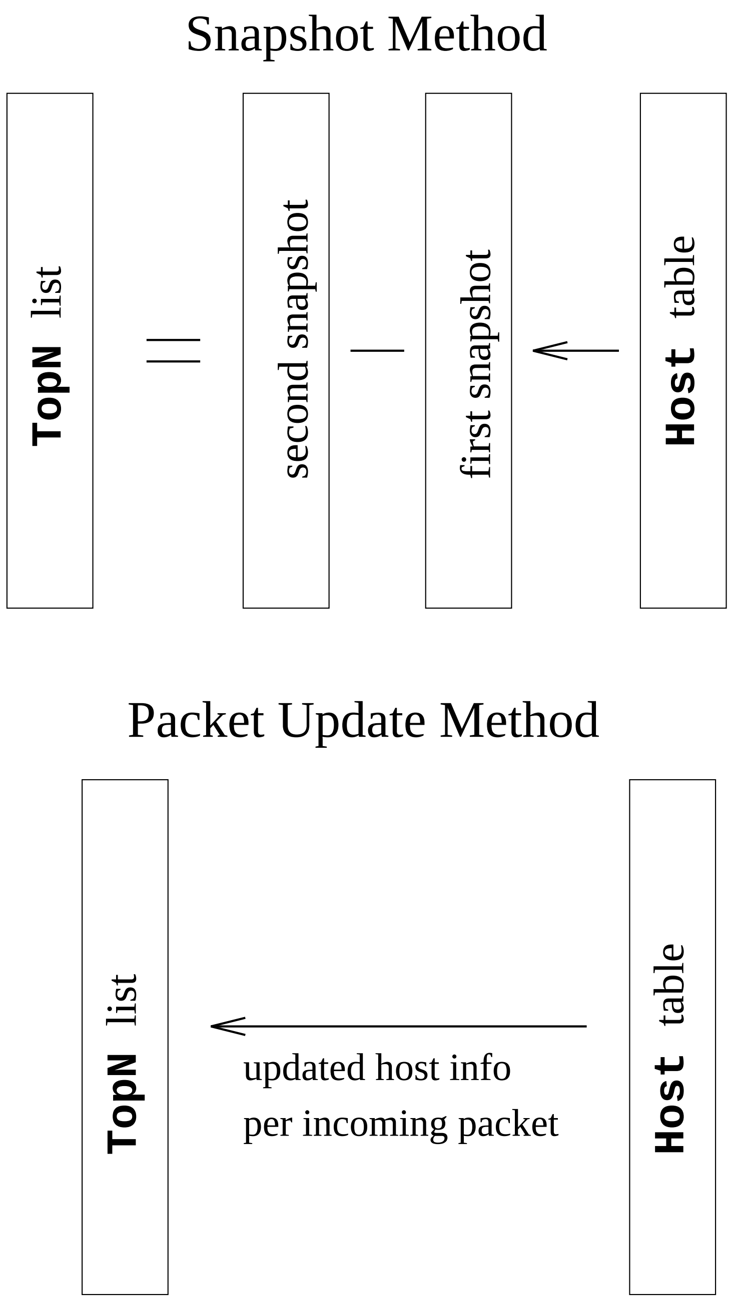

Figure 4.3: The 2 methods for building a `TopN` table

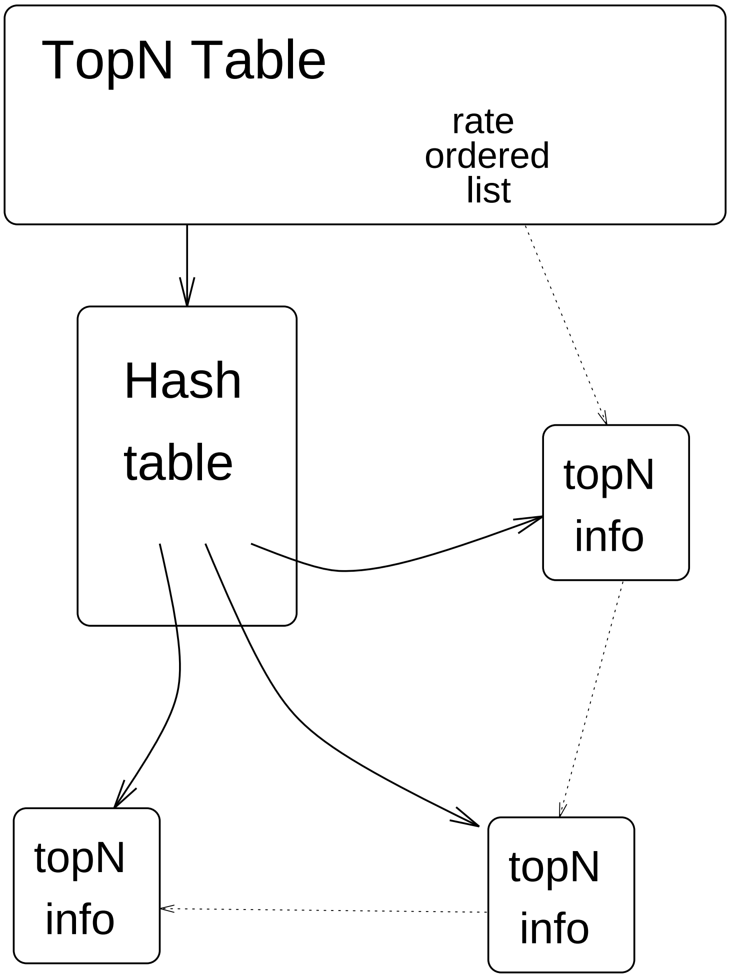

Figure 4.4: `TopN` datastructures

Chapter 5

Row Creation Problems

The problems with default values and row creation were briefly mentioned in section 4.5. Unfortunately, there are more unclear cases when creating RMON rows. Among those are the cases that involve SNMP retries and managers that are accessing the RMON monitor at the same time as other managers.

5.1 SNMP Retries

If an RMON agent reacts slowly and manager applications start sending retries, it can happen that the same request is made to the RMON agent multiple times. This is especially important in the case of row creations and deletions.

Again, table 2.1 can be used to implement a reliable and consistent behavior towards manager applications.

The full state transition diagram (including retries) for setting the `Status` of a collector instance is in figure 5.1. The greatest advantage of this approach is that when a NO ERROR code is returned you can be sure that everything went well. Only if you receive other return codes you may need to check if something really went wrong or SNMP retries occurred.

5.2 Multiple Manager Issues

Some RMON agents will be used by multiple manager applications. Some of the problems that arise with SNMP retries, will also occur in these cases. However, multiple and identical requests are now also made by different managers and not just by one.

Race conditions can occur when more than one manager application is trying to create a new instance in one and the same collector group. To protect against the (mis)use of each others instance it is best that manager applications follow the next procedure:

- Send a SNMP set request with both a set of the `Status` variable (value: `CREATE REQUEST`) and the `Owner` string (value: globally unique name of manager that is performing the request).
- If a manager application receives no errors it can conclude that the operation was successful and it owns the new instance. However, if an `BAD VALUE` error is received, the `Status` and `Owner` values of the wanted instance need to be retrieved again. Are those values what the manager application intended them to be, then apparently a retry has occurred. In this case the application owns it too. However, if the retrieved values are not what they were expected to be, then another manager got the instance first and a new index must be tried.

This procedure relies heavily on the `Status` transition diagram of section 5.1. With another `Status` scheme you may not be able to detect or resolve errors. It must also be possible to create an entry and set the `Owner`in one and the same SNMP request. Some agents may not support this or the order of these requests in one PDU may be significant (which is wrong: the SNMP standard [3] prescribes that order should not be significant).

The sequential approach for choosing new instance indices shows its main disadvantage in this scenario. Every time a manager application needs to go back and choose another index, it has to go through all existing entries (entries may have been deleted in the mean time). With the MIB-next approach, the manager can start at the entry it just tried to obtain. The pseudo-random index method may be cheaper to use.

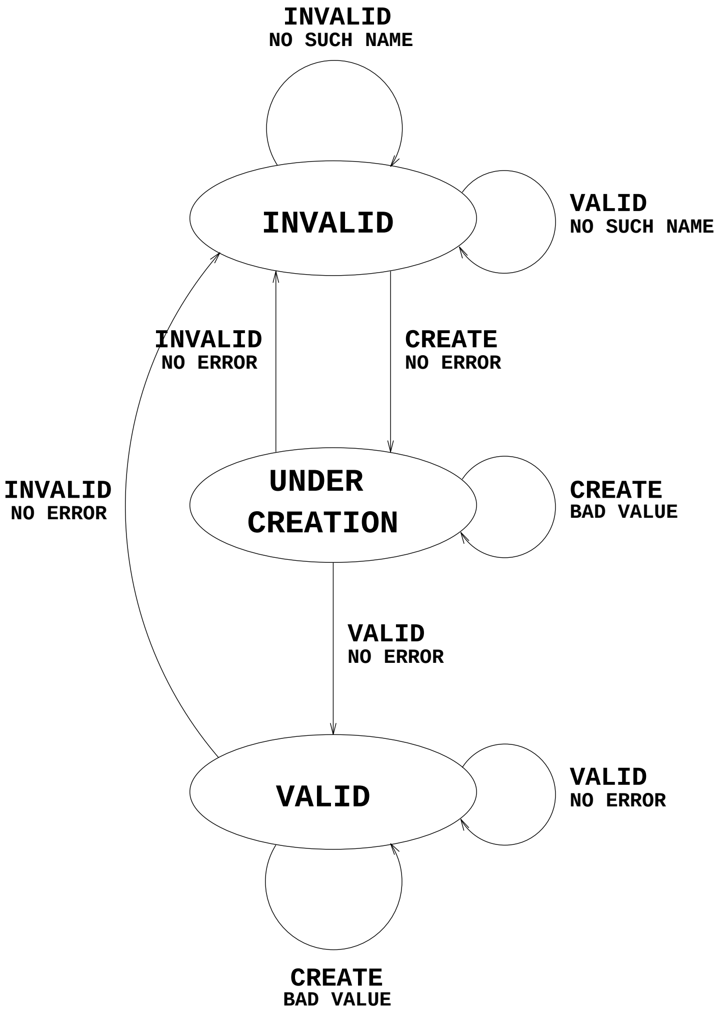

Figure 5.1: `Status` state diagram

Chapter 6

Frequently Asked Questions

This chapter summarizes a number of frequently asked questions. Most of these have already been dealt with in the main text, but are repeated here for easy look-up.

Question What is the unit of PktDataOffset in the `Filter` group?

Answer Bytes.

Question Should filtering be done on the level of bytes?

Answer Yes.

Question Can tables like `Host` and `Matrix` get filtered input instead of the raw packet stream of the interface?

Answer Yes, some implementations allow you to set the `DataSource` of certain groups to the `Channel` index. Only packets that pass the filters attached to this channel will be forwarded to the receiving collector.

Question Can frames that have been reported to be too long or too short, still be considered as frames that have a valid host address?

Answer Yes and no, something is wrong with the frame so it is just another frame with an error in it. But if you find an already existing entry with the same address, it is very unlikely that the host address is wrong. Of course, if the packet is too short, it must still be long enough to contain possible addresses.

Question Should one use error frames for creating entries in the `Host` or `Matrix` collectors?

Answer Never use error frames for **creating** entries.

Question And what to do with CRC align error frames?

Answer Never use these error frames for creating entries or updating statistics in `Host` or `Matrix` entries. Count them only in `EtherStats` groups.

Chapter 7

Conclusions

Bibliography

[1] M.F.B. de Greeve, "Beholder: the next generation", Master's thesis, T.U. Delft, Department of Electrical Engineering, aug 1992.

[2] S. Waldbusser, "Remote network monitoring management information base", RFC 1271, 1991.

[3] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "A simple network management protocol", RFC 1157, 1990.