op5 Trapper 1.0

Administrator’s Manual

Draft v0.1 by A.S. /covers work flow & writing handlers guidelines
Draft v0.2 by R.K. /added the command line management tools description
Draft v0.3 by A.S. /updated the available actions
Draft v0.4 by R.K. updated to reflect handler->traped change

Contents

Preface

Throughout the manual the term `trap` from SNMP v1 and `notification` from SNMP v2 are used interchangeably except the places where it is stated explicitly.

For the sake of readability of the examples we are using the fake, short SNMP OIDs like ".1.1", ".1.2", ".1.3". Of course, you have to use the real OIDs instead.

Workflow

The backbone of trapper are the handlers. Handler - is a rule that describes how SNMP traps and notifications influence nagios services. Handlers are created using a special domain specific language (DSL) described in chapter #2 “Writing Handlers”. Each time the handler is run it may change the state of one service on one host.

Handler can be associated with several OID patterns. Each OID pattern represents the trap OID we want the handler to handle or several OIDs if wild-cards are used:

- `*` - star wild-card matches with any number of any symbols, so that pattern `*` will match any incoming trap and `.1.2*` will match all the traps that start from `.1.2`

There is also a special OID pattern `fallback` to denote fallback handlers. If trap OID does not much any of the OID patterns it will be processed by fallback handlers.

OID patterns are stored in a ordered way. When the new trap arrives, its OID is compared with each OID pattern from first to last and for each match the associated handler is executed. If no match have happened all the fallback handlers are executed one by one. One OID can match several patterns so one trap can change the state of several services.

Lets assume we have the following OID pattern list:

***	*Common*
.1.1	*Temperature*
*.1.**	*Switch trap*
fallback	*Counter*
fallback	*Test*

Then

- *trap `.2.2` will be handler by `Common` handler only;*
- *trap `.1.2` will be handled by `Common` handler, then by `Switch trap` handler;*
- *trap `.1.1` will be handled by `Common` handler, then by `Temperature` handler and then by `Switch trap` handler;*
- *Handlers `Counter` and `Test` will never be used as we have `Common` handler matching all traps.*

If there is no match-any pattern and the list looks like

.1.1	*Temperature*
*.1.**	*Switch trap*
fallback	*Counter*
fallback	*Test*

Then

- *traps `.1.1` and `.1.2` will be handled the same way*
- *trap `2.2` will be handled by `Counter` handler and then by `Test` handler*

.

Writing Handlers

Simple Handler

The most simple handler will look like this:

```
result.state = STATE.WARNING
```

This will set the service `SNMP Traps & Notifications` on the host the trap originated from to state WARNING with empty status information.

These are the valid state values:

- STATE.OK
- STATE.WARNING
- STATE.CRITICAL

-- and they are pretty self explanatory.

Lets assume the device is sending us trap `.1.1` if it is overheat, and trap `.1.2` if temperature is normal again. Then we will match handler

```
result.state = STATE.CRITICAL
```

with oid pattern `.1.1`, and handler

```
result.state = STATE.OK
```

with oid pattern `.1.2`

Customizing Result

If the default values do not feat, they can be customized by setting the appropriate fields of the result structure:

- result.host - the host name
- result.service - the service description
- result.message - the status information

Lets assume that for some testing reasons we have configured nagios with a special host with host name `Testing Host` and service with service description `Testing Service`. And we want trap `.1.1` from any host to set it to critical state. Then we will match handler

```
result.host = “Testing Host”
result.service = “Testing Service”
result.message = “Testing some stuff...”
result.state = STATE.CRITICAL
```

with oid pattern `.1.1`. Note the quotes around the text values.

Tagging Result

More granular service state management is available dividing them on sub-services based on the value of the

- result.tag

field. If not set it has the value `default` and you can set it to any non-empty text value.

The service state is always set to the worst state of all of its tagged sub-services. Thus if service has only one (e.g. default) sub-state then they are equivalent - the sub-service state change will be directly mapped on the service state. But in case of two or more sub-services the state change of one of them may not influence the whole service state immediately.

Lets assume the device is sending us trap `.1.1` if it is overheat, `.2.1` if it is overloaded, trap `.1.2` if temperature is normal again and trap `.2.2` if the load normalized. And we don’t want to have separate service for these events but rather want one `Over...` service that indicates overheat and overload.

We can try to match handler

```
result.service = “Over...”
result.state = STATE.CRITICAL
```

with traps `.1.1` and `.2.1` , and handler

```
result.service = “Over...”
result.state = STATE.OK
```

with traps `.1.2` and `.2.2`.

But this not work if the device is overheat and overload at the same time. Then we will get `.1.1` and `.1.2` in a row and putting the service in critical state. Lets assume then that device load normalized but it is still overheat. Then we will get one trap `.2.2` that will put service in ok state, hiding from us the fact of overheat.

The tagging solves the problem. We can accomplish the task the following way.
Match trap p`.1.1` with

```
result.service = “Over...”
result.tag = “heat”
result.state = STATE.CRITICAL
```

trap `.1.2` with

```
result.service = “Over...”
result.tag = “heat”
result.state = STATE.OK
```

trap `.2.1` with

```
result.service = “Over...”
result.tag = “load”
result.state = STATE.CRITICAL
```

trap `.2.2` with

```
result.service = “Over...”
result.tag = “load”
result.state = STATE.OK
```

Tagging is also used for traps folding. The traps resulted with the same host, service, state and tag are considered duplicates and thus are folded into one trap with appropriate count number.

Getting the Traps Details

If more information about the received trap is needed in handler it can be accessed via the `trap` structure. These

- trap.oid - the oid of the trap
- trap.host - the host that have sent the trap

- are the most general `trap` fields.

Lets says we want the service `SNMP` to became critical on arrival of two traps `.1.1` and `.2.1`, but we also want to know what trap have caused the last state change, then we can write the following handler

```
result.service = “SNMP”
result.state = STATE.CRITICAL
result.message = trap.oid
```

and match it with both traps `.1.1` and `.2.1`.

The time and date of trap are stored in the corresponding fields of `trap` structure, and these fields are structures by themselves:

- trap.date.year
- trap.date.month
- trap.date.day

- trap.time.hour
- trap.time.minute
- trap.time.second

Besides these obvious values, one can also use

- trap.timestamp - contains the trap date and time in Unix time format which is especially useful when calculating time differences
- trap.date.weekday - specifies the number of a day in a week, starting from Sunday (Sunday is 1, Monday is 2 ... Saturday is 7)

Still the most interested part of the `trap` structure is

- trap.fields

that contains all trap's variable bindings in a key-value form (also known as associative array or hash map), with variable binding OID as a key. So that

```
trap.fields[ “.10.10” ]
```

will evaluate to the value of variable binding with oid `.10.10`. Note the quotes in the example: OIDs are strings and all strings must be enclosed on quotes.
An alternative, shorter, syntax to access trap.fields is:

```
trap[[ .10.10 ]]
```

This is the same as trap.fields[“.10.10”]. Note the double square brackets and no quotes!

Lets assume we have a device that monitors the bunch of switches and sends us the trap `.1.1` in case of problems with the IP of the problem switch in `.11.12` variable binding. Surely we want that trap to trigger the state not for monitoring device, but for the switch that has problems. Then we need to match the trap `.1.1` with

```
result.host = trap.fields[ “.11.12” ]
result.state = STATE.CRITICAL
```

Lets assume we have a switch that sends us the trap `.1.1` in case of problems on one of its links, and `.1.2` if the link has recovered. In both cases the links` number is in variable binding `.50.5`. And if the link #3 is facing problems, we don't want the service to be made green by notification that link #7 is Ok.

Then the handler for trap `.1.1` can look like

```
        result.service = “Links”
        result.tag = trap[[.50.5]]
        result.state = STATE.CRITICAL
```

and the mirror handler for `.1.2`

```
        result.service = “Links”
        result.tag = trap[[.50.5]]
        result.state = STATE.OK
```

This way we will create sub-service for each of the links. And according to the way sub-services are handled (see ‘Tagging Result’) the `Links` service will be green only if all of its links sub-services are Ok, and thus traps for different links will not interfere.

Constants & Variables

The following kinds of constants can be used in handlers:

- strings - any text surrounded by quotes (e.g. “Example text...”), if text contains the quote the last has to be escaped with backslash (e.g. “Example of \”Quoted\” text...”)
- numbers - any integer of floating point number (e.g. 0, -7, 100000, 3.14159265, 543.21e8)
- nil - a special value which indicates no value

It is possible to define variables of the same types for convenience, like this

```
msg = “Trap from host”
max = 3
c = 4.5
```

The variables exist in scope of one handler call and do not persist their values between different traps processing sessions.

One can perform certain operations with constants and variables and then assign the result to other variables:

- `=` - assigning, assign value on right side to the variable on left
- `+`, `-`, `*`, `/` - arithmetic operations
- `..` - strings concatenation

Concatenation operation applied to number will convert the latter to string first. Arithmetic operation applied to string will try to convert to number first - the handler will fail if that’s impossible. So

```
x = 10 + 1          -- x = 11
x = “10” .. “1”     -- x = “101”
x = “10” + 1        -- x = 11
x = 10 .. “1”       -- x = “101”
```

```
x = “10” + “1”       -- x = 11
x = 10 .. 1          -- x = “101”
x = 10 + “hm...”   -- error
```

`--` starts a comment - everything after and until the end of line will be ignored.

Usually it is a good practice to use variables with self explaining names for the results of calculations or even variable bindings - that helps readability and maintainability.

Lets assume we have a device that sends us the trap `.1.1` in case of overheat. It also includes the actual temperature in Fahrenheits as `.12.13` variable binding. And we want to form a proper message with temperature converted to Celsius. Then the handler may look like

```
temperature_f = trap.fields[ “.12.13” ]
temperature_c = (temperature_f - 32) * 5 / 9
result.state = STATE.CRITICAL
result.service = “Temperature”
result.message = “Temperature is ” .. temperature_c
```

And of course we need to match it with trap `.1.1`.

Conditionals

Checking some conditions and reacting accordingly may be achieved by the following construct:

```
if <condition> then
        <do something>
end
```

where <do something> can be any number of operations we learned so far, including another if-statement; and <condition> is an expression that can evaluate to `true` or `false` - so called Boolean expression, such as

- `==` - equality, true if values to the both sides are the same
- `~=` - inequality, true if values to the both side are different
- `<`, `>`, `<=`, `>=` - compares numbers, produces errors if used with strings

---

Lets assume we have a device that monitors the bunch of switches. If the switch fails the trap `.1.1` will contain variable binding `.10.10` with its IP. If the monitoring device fails it will send the same `.1.1` trap, but variable binding `.10.10` will contain word `self` instead.
Then the handler may look like this

```
maybe_host = trap.fields[ “.10.10” ]
```

```
if maybe_host ~= “self” then
        result.host = maybe_host
end
result.state = STATE.CRITICAL
```

In this case we will use the default host value unless the variable binding `.10.10` contains something other then word “self”.

Several conditions can be combined into one using the logical operators:

- and - `expression#1 and expression#2` is true only if both expressions are true
- or - `expression#1 or expression#2` is true if at least one of expressions is true
- not - reverses the value of expression - `not expression` is true if expression is false and vice verse

Lets assume we have several devices monitoring the bunch of switches. The old version of device can send trap `.1.1` with variable binding `.10.10` containing the IP of failed switch or the message `self` if the device failed by itself. But in case the new device failed by itself the content of `.10.10` is unpredictable, and one must check the `.10.20` variable binding that indicates the device status, with zero meaning no problems. Then the handler suitable for both versions of device may look like this

```
maybe_host = trap.fields[ “.10.10” ]
if maybe_host ~= “self” and trap.fields[ “.10.20” ] == 0 then
        result.host = maybe_host
end
result.state = STATE.CRITICAL
```

In this case we will use the default host value unless the variable binding `.10.10` contains something other then word “self” or the device status is non-zero, indicating monitoring device error.

Often there is the need to react differently depending on some condition. Then the following form can be used:

```
if <condition> then
        <do something>
else
        <do something else>
end
```

Lets assume that we have a device that always sends us the same trap `.1.1`. But it’s variable binding `.10.10` contains problem description or `OK` if there is no problem.

Then handler can look like this

```
msg = trap.fields[ “.10.10” ]
if msg == “OK” then
    result.state = STATE.OK
else
    result.state = STATE.CRITICAL
    result.message = msg
end
```

If there is more then one condition to check the following form can be used:

```
if <condition> then
    <do something>
elseif <condition #2> then
    <do something another>
elseif <condition #3> then
    <do something even more different>
else
    <do something else>
end
```

with any number of elseif/then blocks.

Lets assume that we have a device that always sends us the same trap `.1.1`, and its actual state is kept in variable binding `.10.10` as one of the following strings: `OK`, `Problems`, `Failure`. Then the handler may look like this

```
st = trap.fields[ “.10.10” ]
if st == “OK” then
    result.state = STATE.OK
elseif st == “Problems” then
    result.state = STATE.WARNING
else
    result.state = STATE.CRITICAL
end
```

Context

Variables used in handler (as well as trap and result structures) exist in scope of one handler call and their values do not persist across different handler calls, even between the calls done while processing the same trap (e.g. when the same handler matched the same trap several times).

If there is the need to persist some values between different trap calls the special associative array `context` can be used like this

```
context[ “name” ] = <value>
```

where `value` can be any value you want to remember and `name` is a string key to access it latter:

```
context[ “num” ] = 15
context[ “txt” ] = “Some text”
context[ “previous oid” ] = trap.oid
context[ “previous .10.10” ] = trap.fields[ “.10.10” ]
context[ “last-set-status” ] = result.status
```

To get the persistent value just refer the context with the same key like this

```
var = context[ “name” ]
```

Also if the value with that key was not set before we will get an empty value, so it is better to use the following form

```
var = context[ “name” ] or <default value>
```

Then if the value with that key was not set before we will get the default value we specified:

```
num = context[ “num” ] or 0
result.message = “Previous message was “ .. ( context[ “txt” ] or “empty” )
prev_set_state = context[ “last-set-status” ] or STATE.OK
```

Note that spaces in names are significant - so that “the name” and “the   name” refer to different values - and a good practice will be to avoid them.

Lets assume we have a device that sends us periodically trap `.1.1` with its temperature in variable binding `.10.10`. The temperature below 40 is considered Ok and above 100 is a problem. In between we want the service to be in WARNING state if it is increasing and in OK state if it is decreasing. Then the handler may look like this

```
result.service = “Temperature”
t = trap.fields[ “.10.10” ]
if t < 40 then
        result.state = STATE.OK
elseif t > 100 then
        result.state = STATE.CRITICAL
else
        prev_t = context[ “temperature” ] or 0
        if t > prev_t then
                result.sate = STATE.WARNING
        else
                result.state = STATE.OK
        end
end
context[ “temperature” ] = t
```

Lets assume we have a device that can send us trap `.1.1`. When this trap is receive we want to turn service “Detections” to WARNING state, but if we get 3 such traps within 2 minutes then to turn the service to CRITICAL state. Then the handler may look like this

```
result.service = “Detections”

time_prev_prev = context[ “prev-prev” ] or 0
time_prev = context[ “prev” ] or 0
time = trap.timestamp

if time - time_prev_prev < 120 then
        result.state = CRITICAL
else
        result.state = WARNING
end

context[ “prev-prev” ] = time_prev
context[ “prev” ] = time
```

Here we retreive the timestamps of 2 previos received traps to calculate the time difference and at the end of handler replace them with actual timestamps.

Actions

Some sophisticated tasks can be achieved using the actions - functions with special abilities like changing the standard trap processing flow etc. Currently available actions are

- action.reject() - stop processing this trap by this and all following handlers and drop it
- action.ignore() - stop processing this trap by this handler and proceed to next one if any
- action.done() - finishes the handler normally, useful to return from deep nested constructions

Note the empty parentheses at the end!

Lets assume that every Friday from 6 to 8 pm the network maintenance procedures are held that result in a flood of problem notifications we are not interested in. In this case *we can take the handler like*

```
if trap.data.weekday = 6 then
    if trap.time.hour > 18 and trap.time.hour <= 20 then
        action.reject()
    end
end
```

match it with `*` pattern and put at the beggining of the hadlers list.

Lets assume the device is sending us the `.1.1` trap in case of any problems. The cause of problem can be retrieved from variable binding `.10.10`. We want to check overheat problem only, all other problems may be checked by other handlers. Then the handler may look like

```
if trap.fields[ “.10.10” ] ~= “Abnormal temperature” then
    action.ignore()
end
result.service = “Temperature”
result.state = STATE.CRITICAL
result.message = “Overheating...”
```

If matched with `.1.1` trap this handler will skip all non-temperature-related traps, still giving other handlers a chance to process the trap for other reasons.

Deep Dive

Handlers DSL is actually created on top of a subset of Lua programming language - a powerful, fast, lightweight, embeddable scripting language. If your need more programming power than covered in this manual, like loops or iterators, feel free to refer Lua Reference Manual. Note, however, that handlers a run in separate secure sandboxes, and potentially dangerous functionality, like access to file system, will not be available.

Using Command Line Tools

Handlers and bindings are managed by a tool called ‘traped’ which is located in /opt/trapper/bin. It takes a command as its first argument, then optionally additional arguments.

Command line syntax::

# traped list modules
prints a list of all modules

# traped list handlers
prints a list of all handlers

# traped list matches
prints a list of matches in a form: <trap_oid> <handler_name>

# traped create module <*name*>
creates an empty module <name>
example:

```
      # traped create module test
```

# traped create handler <*name*>
creates an empty handler <name>
example:

```
      # traped create handler test
```

# traped read <*name*>
prints handler/module <name> to stdout
example:

```
      # traped read test > test.lua
```

# traped update <*name*>
updates handler/module <name> with a script passed via stdin
example:

```
      # traped update test < test.lua
```

# traped delete <*name*>
deletes handler/module <name>

# traped bind *{ <oid>* | *"<pattern>"* | *fallback }* <*name*>
binds handler <name> to trap <oid>,
or to all traps with oids that match <pattern>,
or to all traps that were not processed by any other handler
examples:

```
      # traped bind .1.2.3.4.5 test
      # traped bind ".1.2.3*" test
      # traped bind fallback test
```

note1: * in a pattern means “a substring of any length consisting of any symbols”
note2: you **must** enclose pattern in quotes

# traped unbind *{ <oid>* | *"<pattern>"* | *fallback }* <*name*>
unbinds handler <name>, essentially reverting the same bind command

# traped move up *{ <oid>* | *"<pattern>"* | *fallback }* <*name*>

move match between trap and handler higher in the list, so that it can be processed sooner
example:

```
# traped list matches
.1.2.3*    test1
.1.2.3.4.5 test2
```

(in case a trap with oid .1.2.3.4.5 comes - test1 will be processed first, then test2)

```
# traped move up .1.2.3.4.5 test
# traped list matches
.1.2.3.4.5 test2
.1.2.3*    test1
```

(in case a trap with oid .1.2.3.4.5 comes - test2 will be processed first, then test1)

# traped move down *{ <oid>* | *"<pattern>"* | *fallback } <name>*

move match between trap and handler lower in the list, so that it can be processed later