University of Amsterdam

MSc System and Network Engineering

Research Project 2

Zero-effort Service Monitoring

Author:
Julien Nyczak

Supervisor:
Rick van Rein
ARPA2.net

August 17, 2015

Abstract

Linux distributions are shipped with tools to monitor services/processes. However, the facility to relay this information to a monitoring station does not exist by default. This research paper proves that zero-effort service monitoring is possible through systemd and SNMP. systemd has an overview of installed services on a machine that boots it. This data can be sent over SNMP with the help of an AgentX subagent. The subagent developed during this project communicates names and statuses of all installed systemd services to a local *snmpd* master agent, which in turn relays it to a monitoring station. The paper analyzes a possibility of subagent integration into systemd. This feature would require only a configured *snmpd* master agent running locally to enable the zero-effort service monitoring.

Contents

Acknowledgments

I would like to thank my supervisor Rick van Rein from ARPA2.net for his invaluable help during this research project. His guidance was an incredible asset.

1 Introduction

Linux distributions are delivered with a large amount of packages allowing a machine to work properly. These packages include scripts with a standardized start/stop interface and PID files to check where a daemon should be running. The operating system also provides plenty of information about processes based on their PID such as CPU and memory usage or whether they are running. This can be monitored so that the administrator can be alerted when a package stops working and may lead to a service downtime. Unfortunately, the possibility to pass the information to a monitoring station is not a facility that exists by default. In addition, setting up monitoring usually requires a certain amount of manual work. This research paper focuses on implementing zero-effort service monitoring to monitor services in an automated way, as soon as packages are installed.

Various monitoring protocols exist, from proprietary solutions developed specifically for given monitoring systems, to standardized ones such as the Simple Network Management Protocol (SNMP). The goal of the zero-effort service monitoring is to reduce the configuration phase on the monitored host to sane defaults, with no regards to the distribution flavor (Debian-like or Red-Hat-like). Thus, the technology to send monitored service data must be chosen according to this philosophy and the method to fetch it must use a tool present on every recent Linux distributions.

1.1 Research questions

- How feasible is it to integrate service monitoring in a generic manner for different Linux operating systems (i.e. Red Hat-like and Debian-like)?
- How to relay service status automatically to a monitoring station and be aware of changes?

1.2 Related work

There already exist Linux process monitoring solutions. For instance, the monitoring system Nagios[1] has a plugin that can monitor Linux processes via SNMP [2]. It verifies whether a monitored process is running and also checks its CPU and memory consumption. It uses the Host Resources MIB (Management Information Base) from the RFC 2790 [3] classified as a draft standard. The plugin looks for a process name given as an argument and an *snmpwalk* command is triggered to look for this process on the monitored host. The OID (Object Identifier) 1.3.6.1.2.1.25.4.2.1.2 lists the name of all running processes. If the sought process is not in the retrieved list, Nagios will display a "CRITICAL" status. No subagent is required to query this management information base as it is implemented by default in the Net-SNMP package. The data it covers can be fetched as long as *snmpd* is running on the monitored host and *snmpd.conf* configured to accept queries for this OID. However, the *hrSWRunTable* of this MIB only contains information for running pieces of software installed locally and not all services are taken into account[2]. Moreover, an application specific plugin such as the *check_snmp_process.pl* script is needed in order to know the status of a process if it is not running.

Similarly, *snmpd* can be configured to monitor specific processes thanks to the UCD-SNMP-MIB management information base [4] as it is implemented by default in the Net-SNMP package. The *prTable* of the MIB gives information such as process names, the number of running instances and error flags. The *PROC* directive to be manually inserted[3] in the *snmpd.conf* allows to define which process should be monitored and how many instances should be running. The list of possibly monitored processes can be retrieved from the *ps -ef* Linux command. Once the *snmpd.conf* file is configured as desired, it is possible to verify whether a monitored process is properly running with the help of the *prErrorFlag* column. The error flag is set to 1 if there is an issue, e.g. the number of running instances is too high or too low compared to what is defined in the *snmpd.conf* file for that specific process, or set to 0 otherwise.

These solutions try to fulfill the same goal, i.e. monitor Linux processes.

[1] `https://www.nagios.org/`

[2] But only processes that can be fetched with the *ps -ef* command, kernel threads excluded. Several systemd services are thus missing

[3] One per monitored process

However, they either do not take into account all services installed on a host, or are too fine-grained and necessitate to be meticulously configured or do not advertise the exact status of services. Hence, a new solution must be designed.

2 Tools and protocols

Section 2.1 focuses on the possible tools to collect monitored data. In Section 2.2 are described the possible monitoring protocols for the design of our monitoring solution. Section 2.3 explains the choices that have been made to best attend the purpose of this paper.

2.1 Tools

Here are discussed the tools that can help in fetching process/service information to be sent to a monitoring station.

2.1.1 Kernel tools

Linux comes with lots of command line tools. Some of them help to manage systems and are very handy in scripts because they can be easily parsed. Moreover, they are present in all distributions and thus they are fully valid as options for the design of our monitoring solution.

When it comes to processes/services in the Unix world, two commands come to mind, namely *ps* and *top*. *ps* gives information about running processes. It is often used with options in order to get a broader view of what is running on a system. For instance, the *-ef* option lists every running process along with inter alia their start time and PID. The *-aux* option does the same but with even more details such as CPU and memory usage. *top* displays an interactive real-time view of processes running on a system. It is a very useful command for a system administrator as it can easily highlight what are the most resource consuming processes. It is not the best choice for our needs though, because it is dynamic and not very convenient for scripting. Furthermore, similar information can be found with the *ps* command.

As said earlier, Linux packages often contain a script with a start/stop interface for services. These scripts are located in the */etc/init.d/* directory and are known as init script files, mainly used in operating systems booting the System V init system. The *service* command relies on those scripts and permits among others to know the status of a service, to start or stop it. Even though the *service* command seems to be interesting for the design of our monitoring solution, it is a part of System V, now being replaced by the new init system, systemd. Therefore, it will probably become deprecated in the future.

2.1.2 systemd

systemd has been developed in 2010 by Lennart Poettering [5], a Red Hat engineer. It is a system and service manager for Linux and is inter alia responsible of starting the necessary components at boot time. Its long term goal is to replace other init systems such as System V or Upstart. Packages make an individual choice between the booting system, and most are migrating to systemd with the support of distributions because most of distributions today are shipped with systemd. Ubuntu boots systemd by default only since its last version, i.e. version 15.04 from April 2015 [6].

systemd rethought the way of starting processes. The serialization of the boot-up is dropped and replaced with socket and D-Bus parallelizing [7]. In other words, a process is not obliged to wait for another to be started. It allows a Linux system to boot in a more efficient way which inevitably causes the system to boot faster. In addition, it makes use of Linux control groups (called cgroups) that permits to hierarchically organize processes to fine-tune resource allocation.

Services, sockets, devices, etc. under systemd are described in a declarative language in so-called unit files. They are more structured than the code pieces in init scripts; the extra structure enables to automatically derive from them an extra functionality such as monitoring requirements. Moreover, unit files are compatible with all distributions booting systemd whereas init scripts must be adapted to them[8]. Init scripts are still supported by systemd since some packages lack unit files. The suffix of a unit file defines its type, i.e. a service unit is named "service_name.service". Units are linked to a target which are meant to group units. Units belonging to the same target start at the same time.

systemd is provided with tools to facilitate its management. The *systemctl* command allows to deal with units. Units can be stopped, reloaded, restarted, enabled, etc. Their status can be checked in a general manner with options such as *is-active* (is it running?) or in a more specific way with *is-enabled* (does it start at boot time?). The list of all units can be displayed with no regards to their state. Detailed properties of a unit can also be known with the *show* option.

Although systemd seems to be the next init system, it raised a lot of controversy [9]. Detractors say it is more than a new init system and most importantly, it does not follow the Unix philosophy. Lennart Poettering replied to those attacks in a blog post [10].

2.2 Protocols

Monitored data has to be made available to a monitoring station. The protocol needed for that must involve a minimal configuration process on the monitored host.

2.2.1 Proprietary protocols

Well known monitoring systems tend to develop their own protocol to retrieve monitored data even though they are often able to use standardized technologies such as SNMP, SSH or ICMP. Their proprietary protocols are usually based on the client-server model and necessitate an agent listening on a specific port to be installed and configured on a remote host. One example would be the NRPE (Nagios Remote Plugin Executor) protocol [11] where a host listens for a request coming from a Nagios server, replies to it by executing a plugin and sending back the result the plugin produced. Plugins must be present locally on the remote host. In spite of their practicality, those protocols cannot be chosen for the implementation of our monitoring solution, mainly for three reasons:

- They work only with their monitoring system
- They are not standardized
- They necessitate too much configuration on a remote host (agent and plugin installation, specific user creation)

2.2.2 SSH

SSH (Secure Shell) is a standardized protocol [12] that has been designed to remotely login in a secure way over an insecure network. SSH can also serve for monitoring purposes as commands discussed in Section 2.1 can be remotely executed by scripts and thus data be retrieved. In addition, monitoring systems usually implement SSH monitoring [13] [14]. But monitoring over SSH has several requirements:

- The *openssh-server* package must be installed on the remote host
- A specific user must be created on the remote host with a limited set of commands it can execute, to avoid security issues

- There must be a public key exchange to allow password-less login

The efforts required to setup monitoring over SSH clearly show that this protocol cannot be chosen to send monitored data for our monitoring solution because requirements necessitate too much configuration.

2.2.3 SNMP

SNMP stands for Simple Network Management Protocol [15]. It is a standardized management protocol also meant for monitoring, initially designed for devices running on IP networks, but can be extended to applications.

SNMP is made of three main components, the monitored device or application, an agent that collects the monitored data, and a manager that requests the data (called nework management station or NMS). The agent and the manager communicate via PDUs (Protocol Data Units) transported with UDP.

The information accessible by the agent is defined in MIB (Management Information Base) modules. MIBs also describe the structure of the information by using syntax rules defined in the Structure of Management Information Version 2 (SMIv2), a subset of ASN.1 [16]. It makes SNMP extensible because many OIDs can be uniquely identified.

Tables can be created in MIBs, where rows represent instances and columns attributes (e.g. processes and their status). Smart monitoring systems should be able to iterate over these tables and automatically discover what should be monitored when provided with the MIB to monitor.

Overall, SNMP seems to be the best choice for our monitoring solution, but it is has some shortcomings; the *Net-SNMP* package is not installed by default on Linux distributions, and the master agent has to be configured in order to be queried from the outside. However, those issues can be quickly solved.

2.2.4 The AgentX protocol

With the Internet growth, new MIB modules have been created either to extend the Internet-standard MIB or to answer to needs of private companies. Unfortunately, the SNMP framework is not flexible enough to deal with all these very specific modules. This led to the development of many SNMP subagents with no defined standard and difficult for vendors to cope with.

This is why the Agent Extensibility or AgentX Protocol has been defined in the RFC2741 [17]. Its framework consists of one master agent that communicates SNMP messages but does not have access to management information[4], and of zero or more subagents that do have access to management information but are not aware of SNMP traffic. The purpose is to separate SNMP protocol knowledge from the management information and couple them through a standard interface, the AgentX protocol. It is also worth noting that a subagent reflects the same modular extensibility that the MIB specification mechanisms allow. A subagent is specifically written for a given MIB.

A master agent and a subagent communicate via the AgentX protocol. In this regard, a subagent always starts AgentX sessions. Its role is also to register MIB OIDs with the master agent and to bind them to actual variables after having instantiated managed objects. On the other hand, the master agent accepts AgentX session requests and deals with SNMP messages.

The Net-SNMP package for Linux fully supports the AgentX protocol [18] and provides a library to write subagents in C [19].

2.3 Choices for the implementation

According to Section 2.1, the *ps* command and *systemctl* from systemd seem to be good candidates for collecting data for our monitoring solution. The *service* command is discarded because *systemctl* provides the same functionality, and even more. Besides, it has been designed for init scripts that are likely meant to disappear with time.

ps is present in all distributions, and *systemctl* in most of them for now, and probably in all in a near future. Nonetheless, *ps* only gives information about running processes. If a service is not running, it will not be listed in the command output. In addition, it covers more than services; some processes may not be worth monitoring, such as commands that can be launched locally by a system administrator. Finally, *ps* is also not aware whether a service is started at boot time where *systemctl* is. The *systemctl* command will thus be the selected tool to poll service information for this project.

The zero-effort service monitoring is meant to work on all Linux distributions. For its implementation, the monitoring protocol must be thus a

[4]Unlike within the SNMP framework where the master agent does have access to management information

standard requiring a minimal amount of configuration on a monitored host and it has to be dynamic enough to cope with changes. From what is described in Section 2.2, SNMP coupled with the AgentX protocol appears to be the most suitable. Regardless of the monitored host, a MIB designed for monitoring services will be fed with the exact number of installed services, in a fully dynamic way: if a new service is installed, it will be taken into account, and if it is removed, it will stop being monitored.

Now that we know how to implement the zero-effort service monitoring, a MIB has to be selected and subagent to be developed in order to create a proof of concept showing the feasibility of this project.

3 Approach and methods

The purpose of this section is to present the approach followed that led to the design of the subagent. The subagent runs on two machines with different Linux operating systems: a Debian-like (Ubuntu Desktop 15.04) and a Red-Hat-like (Fedora Server 22). The goal is to demonstrate the universality of the zero-effort service monitoring idea on OSs booting systemd.

3.1 Querying systemd services

systemd offers to query services through the *systemctl* command. It accepts various options but only a few do output what we want to achieve, i.e. retrieving all installed services on a system along with their intended and actual status. The subagent thus polls information using external commands to get service data in order to show the feasibility of the zero-effort service monitoring. However, integrating this concept into systemd as proposed in Section 7, would probably allow the subagent to directly process instant awareness of the changes in a system.

3.1.1 Retrieving the list of installed services

- *systemctl -t service* displays all running services. The "running" word is important here, as services that are in an "inactive" status are not listed. Hence, this command cannot be used by the subagent to poll information from systemd as it lacks data.

- *systemctl -a -t service* retrieves all services handled by systemd regardless of the status ("active" or "inactive") even though they are started with an init script file. However, such services are not taken into account by the command as soon as they are stopped. For instance, on Ubuntu 15.04, the Apache2 package does not have a unit file but is rather still started with an old init shell script. This is the reason why it disappears from the command output as soon as it stops running.

- *systemctl list-unit-files -t service* shows all service units installed on a system that are fully handled by systemd. A service must have a unit file to be in the output of this command which will probably be the case for all packages in a near future.

The *systemctl list-unit-files -t service* command seems to be the most suitable way to retrieve the list of systemd services to be monitored because it displays all service units regardless their status. The fact that non-native systemd packages are not taken into account is a minor issue today that will disappear tomorrow.

3.1.2 Retrieving the status of services

The selected command above does not show much information regarding installed units. It merely displays two columns, their name along with their state. The state does not specify whether the unit is running but rather if it is enabled.

Nonetheless, the *systemctl show unit_name -p ActiveState* command can supply this information and that is why the subagent queries each unit with it. The *systemctl is-active* command is buggy and often cannot retrieve the actual status which is translated with an "unknown" output.

In addition, the subagent runs the *systemctl is-enabled* command to know whether a unit is started at boot time as it outputs only one word and thus, is easier to parse.

3.2 Writing the AgentX subagent

3.2.1 The subagent

Although the Net-SNMP package is provided with a library to write AgentX subagents in C, the subagent has been developed in Python due to the limited expertise in C programming. The Python module called netsnmpagent [20] written by Pieter Hollants and licensed under GNU General Public License version 3 offers such a functionality. Three files are needed to run a subagent:

- a management information base used by the subagent
- the Python subagent itself
- a shell script to start the subagent

The shell script is also in charge of verifying whether the Net-SNMP package is installed and of starting an *snmpd* master agent with which the Python subagent communicates through the AgentX protocol.

3.2.2 The MIB

A subagent has to be written for a specific MIB as described in Section 2.2.4. I have chosen the Network Services Monitoring MIB [21] since it is adapted to the needs of systemd unit monitoring and most importantly, it is standardized. The table it contains that we are interested in is the *applTable* under the OID 1.3.6.1.2.1.27.1. Three of its columns are adequate: *applIndex*, *applName* and *applOperStatus*. The index uniquely identifies the service unit and the name is the name of the service unit. Even though indexes do not add any particular value, they are required by the netsnmpagent Python module. Nevertheless, I believe that names would suffice.

The operational status column proposes six different statuses: up (1), down (2), halted (3), congested (4), restarting (5) and quiescing (6). However, according to their description in the MIB and from what can be interpreted by their names from an administrator point of view, only the first three and the sixth one can be adopted by the subagent. Indeed, a service is restarted too quickly for the monitoring to detect a restarting status. Moreover, systemd does not have a way of finding out whether a service is congested. Section 4.1.2 explains how these four operational statuses are interpreted by the subagent.

4 Results

This section explains how the proof of concept that illustrates the zero-effort service monitoring idea is constructed. Several items are involved on the monitored host. The monitoring station runs Nagios and asks for service unit's data via SNMP. Figure 1 shows the global workflow.

The source code of the proof of concept can be found on the Github repository of ARPA2[5].

4.1 The Python subagent

Here is described how the Python subagent has been built. I mention first the fine-tune feature which is an override of the default behavior, so that the reader can have a better understanding of the algorithm part. Furthermore, the reader should bear in mind that the "state" of a unit means whether it is "enabled" (started at boot time) and "status" whether it is "active" (running). The operational status is the status of a unit retrieved via SNMP.

4.1.1 Fine-tuned monitoring

The zero-effort service monitoring idea suggests to monitor every service by default in order to have an overall overview of all installed services regardless of their state. The subagent must transpose this behavior.

However, this means a lot of information as the number of installed service unit files is about two hundred by default. This can be difficult to cope with from an administrator point of view, especially because a unit that is "down" or "halted" does not necessarily mean that something is wrong with it. Similarly, a unit that is "up" may cause problems to an administrator. Hence, one should have the possibility to fine-tune how the status for each unit is translated by the subagent.

To that end, the agent takes into account several files that can contain the names of installed service units. It is up to the administrator to fill them in, they are empty by default. Like *systemctl* commands, files are called each time the subagent wants to update the data (thirty seconds by default). It means that the data retrieved over SNMP may not be accurate if something changed between two updates. However, the data update frequency can be set as desired.

[5] `https://github.com/arpa2/sytemd-snmp-zeroconf`

The file called *not_monitored_units* is meant for units that need to be discarded from the monitoring, to shorten the number of monitored items for instance[6]. The *enabled_units* file allows the subagent to know which service must be enabled so that it can translate the status of the unit according to what systemd reports, i.e. if a unit should be enabled but it is not, the subagent changes its status. One may also like to monitor service units that must be down. The *units_to_be_down* file permits the subagent to verify this.

All this configuration process may seem to interfere with the zero-effort service monitoring philosophy, but it is not. The full unit service monitoring is set by default. The fine-tune feature has been implemented only for administrators who wish to use it.

4.1.2 The algorithm

First of all, an instance of the netsnmpagent class is created where the path to the NETWORK-SERVICES-MIB is defined.

Subsequently, the table is created with a class that takes at least three arguments which are the OID string of the *applTable* in the MIB, the indexes and the columns[7] all identified by their OID.

The algorithm can now start the data update. The first task is to store into a list all service units made available by the *systemctl* command[8] and to discard units present in the *not_monitored_units* file and those starting with a "-" or containing a "@". Units such as *-.slice* or *systemd-rfkill@.service* cannot be queried with other *systemctl* commands. This is a bug that needs to be reported to systemd people. Next, the status of monitored units is inserted into another list[9]. Both lists are merged into a dictionary with unit names as keys and statuses as values. A third list is created that contains units from the *units_to_be_down* file so that the algorithm can invert the status of those units[10]. The subagent will report an "up" operational status for a unit that is "inactive"[11] and part of the *units_to_be_down* file. As seen in Section 3.2.2, the MIB offers only six operational statuses and none would fit a "Down but OK". Such a unit being declared as "up" by the subagent seems

[6] This can be compared to blacklisting

[7] *applName* and *applOperStatus* columns

[8] *systemctl list-unit-files -t service*

[9] The status is retrieved with the *systemctl show unit_name -p ActiveState* command

[10] "active" to "inactive" and "inactive" to "active"

[11] And vice versa

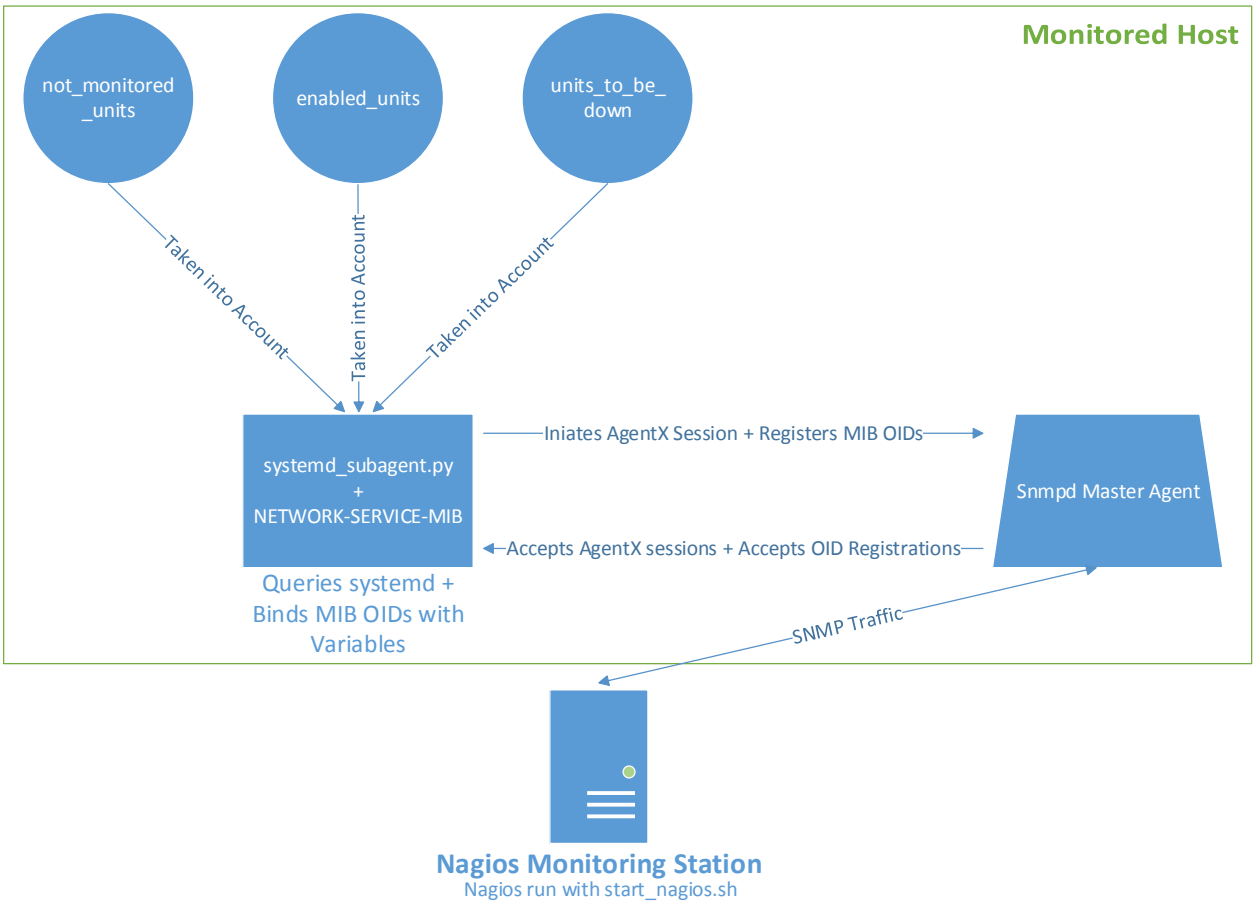

Figure 1: The workflow of the proof of concept

to be the closest operational status that reflects the reality, i.e. everything is running correctly. Then, a fourth list is created, this time to host units added to the *enabled_units* file. The first list and the fourth one are put together to form a second dictionary. Thanks to this, the algorithm can now iterate over the fourth list to change the status of a unit from the first dictionary to "disabled" (if a unit should be enabled and it is actually not).

Finally, indexes, rows and cells are created with the *index*, *addRow* and *setRowCell* functions, respectively. Table 1 shows how the subagent translates statuses retrieved with the *systemctl* commands to the operational sta-

[12]The unit is present in the *enabled_units* file and its status has been swapped from "active" to "disabled" by the algorithm

systemd unit status	SNMP operational status	Interpretation
active	up (1)	The unit is "active", or "inactive" and supposed to be
disabled[12]	halted (3)	The unit should be enabled and is not
inactive and enabled	quiescing (6)	The unit will be "active" after the next reboot
inactive or failed	down (2)	The unit is "inactive", or "active" and not supposed to be

Table 1: Translation of systemd unit status to SNMP operational status

tuses made available through SNMP.

4.2 Monitoring with Nagios

SNMP is used for monitoring purposes but it is not very handy for observing quickly what is happening with the monitored items. Hence, the data that can be fetched with SNMP may be relayed to a monitoring station hosting a graphical web application, easy to read. Writing an application specific-plugin was not the focus of this paper, therefore the Nagios implementation is more a workaround than a long term solution and may seem rudimentary.

Nagios is the chosen monitoring system that is running on the monitoring machine for this proof of concept. I was looking for a Nagios plugin satisfying my requirements, i.e. one that can via SNMP iterate over tables and retrieve all names and operational statuses of units. In principle, it should be possible for an SNMPc plugin to automatically scan an attached network to automatically discover the zero-effort service monitoring on each host, and to automatically add them to the monitored information base.

Unfortunately, I could not find anything fully achieving this goal. The *check_snmp_table.pl* [22] plugin can join two columns of a same table together, e.g. a service unit name with its operational status, with the help of an *snmpwalk* command. It is published under the GNU General Public License v2.

However, statuses in this plugin are thresholds, i.e. a "CRITICAL" value cannot be between an "OK" and a "WARNING" one. This does not fit what the NETWORK-SERVICES-MIB offers where the "up" status is 1, "down" is 2, and "halted" is 3. Thus, the perl script has been modified to allow the wanted behavior.

Furthermore, the plugin does not offer an automatic iteration over tables. It is called in a Nagios configuration file for a specific host for a specific OID. A lot of configuration is required as the subagent retrieves hundreds of OIDs

SNMP operational status	Nagios interpretation
up (1)	OK
halted (3)	WARNING
quiesing (6)	WARNING
down (2)	CRITICAL

Table 2: Nagios interpretation of SNMP operational status

by default. To solve this problem, I have written a bash script that first performs an *snmpwalk* to retrieve the names of all monitored service units on a remote host, and creates a Nagios configuration file accordingly for that host. Table 2 shows how Nagios interprets the operational statuses retrieved via SNMP.

The reader should keep in mind that the script has been developed only for illustration purposes. It has to be run each time the number of monitored units changes on the remote host, when units are added or removed from the *not_monitored_units* file for example. In addition, an *snmpwalk* is triggered whenever Nagios performs a check because the configuration file created with the bash script will call *check_snmp_table.pl* for each monitored service units. This can be very CPU consuming for the monitored host. A solution would be to run an *snmpwalk* command every time Nagios performs a check, to cache the output and use the cache to join unit names with operational statuses.

Host	Service	Status	Last Check	Duration	Attempt	Status Information
fedora_RP2	NetworkManager-dispatcher.service	WARNING	07-08-2015 00:37:01	1d 5h 24m 57s	1/1	WARNING - NetworkManager-dispatcher.service is 6 > 2
	NetworkManager-wait-online.service	CRITICAL	07-08-2015 00:36:59	1d 5h 24m 57s	1/1	CRITICAL - NetworkManager-wait-online.service is 2 = 2
	NetworkManager.service	OK	07-08-2015 00:36:58	0d 5h 39m 57s	1/1	OK - NetworkManager.service is 1
	abrt-ccpp.service	OK	07-08-2015 00:37:02	0d 5h 39m 57s	1/1	OK - abrt-ccpp.service is 1
	abrt-journal-core.service	CRITICAL	07-08-2015 00:37:03	0d 5h 49m 57s	1/1	CRITICAL - abrt-journal-core.service is 2 = 2
	abrt-oops.service	OK	07-08-2015 00:37:00	0d 5h 39m 57s	1/1	OK - abrt-oops.service is 1

Figure 2: Zero-effort service monitoring with Nagios

This is an example of how the zero-effort service monitoring can be implemented in a monitoring system such as Nagios which is illustrated by Figure 2. It is essential to the reported work that nothing Nagios-related has been installed on the remote host. This demonstrates exactly the strength of SNMP, to be a monitoring standard.

5 Operating system integration

So far, a subagent built upon the Net-SNMP package has been developed to monitor systemd service units. The universality of systemd allows the subagent to be run on any operating system that boots systemd. However, there is only one requirement for packages to be handled by the subagent. They must be delivered with a systemd service unit file.

The challenge now is to integrate the subagent within an operating system so that the zero-effort service monitoring idea can be fully achieved. This can be approached in several ways.

Assuming that the subagent is written in C, one solution would be to package the subagent into an *rpm* or *deb* file with the Net-SNMP package as its dependency. Once the package installed, systemd could start the *snmpd* service at boot time. The subagent would have thus a master agent running to connect to and make service monitoring available.

A second possibility would be to dynamically extend systemd by integrating the C subagent as a shared library (i.e. a *.so* file). Thus, the functions of the subagent would not have to poll service information with *systemctl*, but they could rather deal directly with systemd. *.so* files are convenient in the sense that they can be installed from a separate package and still can modify the core package, i.e. systemd in this case. If the Net-SNMP package would be installed on the remote host, the intended functionality of the shared library (i.e. service monitoring) would be loaded, or skipped in the opposite case.

Another approach would be to integrate directly the subagent within the Net-SNMP package. Similarly to the Host Resources MIB discussed in Section 1.2, systemd service monitoring would be enabled by just having the Net-SNMP package installed. But it implies that the Net-SNMP team agrees on that. In addition, the work that has been done during this project would be then labeled under Net-SNMP and not ARPA2.net.

The first two approaches are the most suitable with a preference for the second one as it would be directly linked to systemd. Only the Net-SNMP package would have to be additionally installed, and *snmpd.conf* to be quickly configured to allow SNMP queries coming from a monitoring station. Enabling *snmpd.service* with systemd will make it start at boot time which would result in activating service monitoring via SNMP.

6 Conclusion

systemd makes the zero-effort service monitoring idea possible. Regardless of whether it is running on a Debian-like or Red-Hat-like operating system, polling services and their status is achievable with the same commands as long as they do have a systemd service unit file. Hence, the AgentX subagent specially developed for systemd service monitoring during this project can be run on various OSs booting systemd. It provides monitoring of each service by default with no need for configuration[13] and can be fine-tuned if requested.

The Network Services Monitoring MIB the subagent has been written for may show some limitations though. This is particularly true when fine-tune monitoring is set[14]. Two other operational statuses would have made things clearer from an SNMP point of view:

- "Down but OK"
- "Up but not OK"

Currently, we are stuck with "up" and "down" operational statuses. This can be confusing when a service unit is "inactive" and reported by the agent as "up" just because the unit is part of the *to_be_down_units* file. One could think of using the "halted" operational status instead. But it is already used for units that should be enabled and are not.

In order to fully achieve the zero-effort service monitoring idea, the subagent must be integrated with the operating system. This can be done by directly integrating it into systemd as a shared library. The Net-SNMP package would be a required dependency and the *snmpd.service* enabled in systemd.

Only an *snmpd* daemon that can be queried on the monitored host is required for any respectable monitoring application[15] to retrieve service data. This is the reason why the whole idea of this project is based on SNMP, to minimize the configuration process. However, the monitoring system must have a way to deal with SNMP tables in an automated way.

[13] From a systemd point of view. The subagent just needs an *snmpd* process running
[14] Especially when the *to_be_down_units* file is filled in
[15] An application than can handle SNMP

7 Future Work

In order to fully integrate the zero-effort service monitoring, we have seen in Section 5 that the proof of concept could either be packaged into a *deb*/*rpm* file or even integrated within systemd. The actual state of proof of concept of the subagent does not allow that for both solutions. Hence, it should be re-written in C.

Finally, new operating statuses in the MIB would sweep away any ambiguities. At least two new operational statuses could be added as discussed in Section 6. With standard but clear names so that they can apply to network services as well, one could think of extending the RFC 2788 instead of writing a new MIB. The MIB is standardized and it would be regrettable not to take advantage of this.

8 References

[1] J. Case, M. Fedor, M. Schoffstall & J. Davin, *A Simple Network Management Protocol (SNMP)*, May 1990, https://www.ietf.org/rfc/rfc1157.txt

[2] Patrick Proy, *Nagios plugin for process monitoring*, June 2007, http://nagios.proy.org/snmp_process.html

[3] S. Waldbusser & P. Grillo, *Host Resources MIB*, March 2000, https://tools.ietf.org/html/rfc2790

[4] Wes Hardakerr, *UCD-SNMP-MIB*, January 2009, http://www.net-snmp.org/docs/mibs/ucdavis.html

[5] Wikipedia, *Lennart Poettering*, May 2015, https://en.wikipedia.org/wiki/Lennart_Poettering

[6] Wikipedia, *Adoption of systemd*, June 2015, https://en.wikipedia.org/wiki/Systemd#Adoption_and_reception

[7] freedesktop.org, *systemd System and Service Manager*, June 2015, http://www.freedesktop.org/wiki/Software/systemd/

[8] freedesktop.org, *systemd for Administrators, Part III*, October 2010, http://0pointer.de/blog/projects/systemd-for-admins-3.html

[9] Chris Hoffman, *Meet systemd, the controversial project taking over a Linux distro near you*, October 2014, http://www.pcworld.com/article/2841873/meet-systemd-the-controversial-project-taking-over-a-linux-distro-near-you.html

[10] Lennart Poettering, *The Biggest Myths*, January 2013, http://0pointer.de/blog/projects/the-biggest-myths.html

[11] egalstad, *NRPE - Nagios Remote Plugin Executor*, September 2013, https://exchange.nagios.org/directory/Addons/Monitoring-Agents/NRPE--2D-Nagios-Remote-Plugin-Executor/details

[12] T. Ylonen & C. Lonvick, *The Secure Shell (SSH) Protocol Architecture*, January 2006, https://tools.ietf.org/html/rfc4251

[13] Zabbix LLC., *Agentless Monitoring*, August 2015, http://www.zabbix.com/agentless_monitoring.php

[14] Nagios Enterprises, LLC., *Nagios XI Monitoring Hosts Using SSH*, August 2012, https://assets.nagios.com/downloads/nagiosxi/docs/Monitoring_Hosts_Using_SSH.pdf

[15] J. Case, M. Fedor, M. Schoffstall & J. Davin, *A Simple Network Management Protocol (SNMP)*, May 1990, https://www.ietf.org/rfc/rfc1157.txt

[16] K. McCloghrie, D. Perkins & J. Schoenwaelder, *Structure of Management Information Version 2 (SMIv2)*, April 1999, https://tools.ietf.org/html/rfc2578

[17] M. Daniele, B. Wijnen, M. Ellison & D. Francisco, *Agent Extensibility (AgentX) Protocol Version 1*, January 2000, https://www.ietf.org/rfc/rfc2741.txt

[18] Dave from the Net-SNMP community, *README.agentx*, May 2011, http://www.net-snmp.org/docs/README.agentx.html

[19] The Net-SNMP community, *SNMP agent API*, April 2001, http://www.net-snmp.org/docs/man/snmp_agent_api.html

[20] Pieter Hollants, *Writing Net-SNMP (AgentX) subagents in Python*, June 2015, https://pypi.python.org/pypi/netsnmpagent

[21] N. Freed & S. Kille, *Network Services Monitoring MIB*, March 2000, https://tools.ietf.org/html/rfc2788

[22] William Leibzon, *SNMP Nagios Plugin*, October 2006, http://wleibzon.bol.ucla.edu/nagios/plugins/check_snmp_table.pl