



TRABAJO FIN DE GRADO
INGENIERÍA EN INFORMÁTICA

Análisis y monitorización de aplicaciones a través de plugins con Naemon

Autor

Sofía Fernández Moreno

Directores

Alberto Guillén Perales

Héctor Pomares Cintas



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

Granada, 5 de septiembre de 2019

Análisis y monitorización de aplicaciones a través de plugins con Naemon

Autor

Sofía Fernández Moreno

Directores

Alberto Guillén Perales

Héctor Pomares Cintas

Análisis y monitorización de aplicaciones a través de plugins con Naemon.

Sofía Fernández Moreno

Palabras clave: monitorización , plugin, naemon.

Resumen

El siguiente proyecto propone el despliegue de la aplicación de monitorización Naemon así como el desarrollo de plugins específicos para analizar el uso de elementos concretos de la aplicación. Teniendo en cuenta el histórico recogido por la aplicación de monitorización también se propondrá un sistema de análisis que permita la prevención de alertas o detección de anomalías.

Analysis and monitoring of applications through plugins with Naemon.

Sofía Fernández Moreno

Keywords: monitoring, plugin, naemon

Abstract

The following project proposes the deployment of the Naemon monitoring application as well as the development of specific plugins to analyze the use of specific elements of the application. Taking into account the historical data collected by the monitoring application, an analysis system that allows the prevention of alerts or anomalies detection will also be proposed.

Yo, **Sofía Fernández Moreno**, alumna de la titulación Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 15513804M, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Sofía Fernández Moreno

Granada, 5 de septiembre de 2019

D. **Alberto Guillén Perales**, Profesor del Departamento de Arquitectura y Tecnología de Computadores de la Universidad de Granada.

D. **Héctor Pomares Cintas**, Profesor del Departamento de Arquitectura y Tecnología de Computadores de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado *Análisis y monitorización de aplicaciones a través de plugins con Naemon.*, ha sido realizado bajo su supervisión por **Sofía Fernández Moreno**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 5 de septiembre de 2019.

Los directores:

Alberto Guillén Perales

Héctor Pomares Cintas

Agradecimientos

Muchas gracias a mi familia por el apoyo que me han ofrecido a lo largo de estos años de carrera. A mis tutores, profesores y amigos, tanto los de toda la vida como los que he ido encontrando a lo largo de esta etapa, que gracias a todos ellos me han ayudado a llegar a donde estoy a día de hoy.

Índice general

1	Introducción	17
1.1	Objetivos	17
1.2	Planificación del proyecto	18
2	Estado del arte	19
2.1	Monitorización	19
2.1.1	Ventajas y desventajas de SNMP	20
2.2	Comparación herramientas de monitorización OpenSource . .	21
2.2.1	Zabbix	22
2.2.2	Cacti	23
2.2.3	Pandora FMS	24
2.2.4	Nagios	26
2.2.5	Naemon	27
2.3	Comparativa de Naemon	28
2.3.1	A nivel sistema operativo	28
2.3.2	A nivel aplicación	28
2.3.3	A nivel extensiones	29
2.4	Introducción de Naemon	29
2.4.1	Funcionamiento Naemon	29
2.4.1.1	Archivo de configuración principal	31
2.4.1.2	Objetos	32
2.4.1.2.1	Hosts	32
2.4.1.2.2	Servicios	36
2.4.1.2.3	Comandos	40
2.4.1.2.4	Contactos	41
2.4.1.2.5	Time Periods	42
2.4.1.2.6	Plantillas	42
2.4.1.3	Thruk	42
2.4.1.4	Plugins	44
3	Despliegue de Naemon	47
3.1	Entorno del despliegue	47
3.1.1	Alternativas a utilizar	47
3.1.1.1	Entorno nativo	48

3.1.1.2	Entorno virtual o máquina virtual	48
3.1.1.3	Aplicación de contenedores	49
3.1.1.3.1	Docker Engine	52
3.1.1.3.2	OpenShift	53
3.1.1.3.3	CoreOS rkt (rocket)	54
3.1.1.3.4	LXC (Linux Containers)	55
3.1.2	Alternativa elegida: Uso de contenedores con Docker .	56
3.2	Desarrollo del despliegue de Naemon	56
3.3	Creación de Dockerfile	59
3.3.1	Instalación de imagen base phusion	61
3.3.2	Instalación paquetes	62
3.3.3	Instalación clave GPG	63
3.3.4	Instalación repositorio	63
3.3.5	Inicialización de directorios	63
3.3.5.1	Creación script de inicialización: init.bash . .	64
3.3.5.2	Copia de archivos	65
3.3.5.3	Asignación de permisos	65
3.3.6	Cargar datos en carpeta raíz	66
3.3.7	Protocolo de salida	66
3.3.8	Creación de imagen y ejecución de Dockerfile	67
3.3.8.1	Creación de ENTRYPOINT	67
3.3.8.1.1	Fichero bash de ejecución: run.bash	68
3.3.8.2	Creación de imagen a través de docker build	70
3.3.8.3	Ejecución de imagen	71
3.4	Orquestación de aplicaciones	72
3.4.1	Uso de Docker Compose	74
4	Pruebas de carga	77
4.1	Introducción	77
4.2	Tipos de pruebas de rendimiento	78
4.2.1	Prueba de carga	78
4.2.2	Prueba de estrés	78
4.2.3	Prueba de estabilidad (Soak Testing)	78
4.2.4	Prueba de pico (Spike Testing)	79
4.3	Tipo de prueba seleccionada	79
4.3.0.1	Gatling	79
4.3.0.2	Locust	80
4.3.0.3	Jmeter	81
4.3.1	Alternativa elegida: Locust	81
4.4	Instalación de Locust	82
4.5	Funcionamiento de herramienta Locust	82
4.6	Ejecución distribuida	82
4.7	Configuración Locustfile	85

5	Pruebas de carga en sistema	87
5.1	Despliegue de sistema a través del contenedor Docker	87
5.2	Enlazado con Locust	90
5.2.1	Creación de archivo Locustfile	91
5.3	Pruebas de carga con sistema WordPress	92
5.4	Pruebas de monitorización en Naemon	94
5.4.1	Definición del host	97
5.4.1.1	Host utilizado en la prueba de monitorización del sistema	100
5.4.2	Definición de los servicios	101
5.4.2.1	Servicios utilizados en la prueba de monitorización del sistema	104
5.4.2.1.1	Servicio PING	105
5.4.2.1.2	Servicio para comprobar usuarios actuales registrados	106
5.4.2.1.3	Servicio para comprobar total de procesos ejecutados	107
5.4.2.1.4	Servicio para comprobar la carga actual	108
5.4.2.1.5	Servicio para comprobar el uso de intercambio	109
5.4.2.1.6	Servicio para comprobar el protocolo SSH	110
5.4.2.1.7	Servicio para comprobar el protocolo HTTP	112
5.5	Puesta en práctica: creación de complemento o plugin	113
6	Modelado	119
6.1	Introducción a la carga de trabajo	119
6.1.1	Características de un modelo de carga	120
6.1.2	Selección de la carga de trabajo	121
6.2	Obtención de datos de rendimiento a través de PNP4Nagios	122
6.2.1	¿Qué es PNP4NAGIOS?	122
6.2.2	Instalación de PNP4Nagios a través de Docker	124
6.2.2.1	Integración de PNP4Nagios en Naemon	125
6.2.2.2	Integración de PNP4Nagios en Thruk	127
6.2.3	Exportación de los datos a formato CSV	129
7	Conclusiones y trabajos futuros.	133
7.1	Conclusión.	133
7.2	Trabajos futuros.	134
8	Anexo	135

Índice de tablas

8.1	Tabla HTTP	136
8.2	Tabla PING	137

Índice de figuras

2.1	Zabbix	22
2.2	Cacti	23
2.3	Pandora FMS	24
2.4	Nagios	26
2.5	Naemon	27
2.6	Comparativa a nivel de sistema	28
2.7	Comparativa a nivel de aplicación	28
2.8	Comparativa a nivel de extensiones	29
2.9	Configuración de Naemon[13]	31
2.10	Formato de definición de un objeto de tipo Host	33
2.11	Formato de definición de un objeto de tipo Servicios	37
2.12	Formato de definición de un objeto de tipo Comandos	40
2.13	Formato de definición de un objeto de tipo Contacto	41
2.14	Formato de definición de un objeto de tipo Time-Period	42
2.15	Estructura de conexión de Thruk	43
2.16	Funcionamiento de los plugins en Naemon	45
3.1	Arquitectura de un entorno virtual	49
3.2	Estructura de un entorno con contenedores	50
3.3	Docker	52
3.4	OpenShift	53
3.5	CoreOS	54
3.6	LXC	55

3.7	Estructura del contenedor Docker con Naemon y Ubuntu de forma minimalista	58
3.8	Estructura de un fichero Dockerfile	60
3.9	Comparativas de estadísticas	61
3.10	Ejecución de Docker	72
3.11	Logo de YAML	73
3.12	Ejecución de Docker Compose	75
3.13	Ejecución de Thruk mediante Docker Compose	76
4.1	Gatling	79
4.2	Locust	80
4.3	JMeter	81
5.1	Instalación de WordPress	89
5.2	Pantalla principal Blog de WordPress	89
5.3	Interfaz Locust	92
5.4	Realización de tareas concurrentes	93
5.5	Pruebas de tareas concurrentes	93
5.6	Gráfico generado en las ejecuciones	94
5.7	Contenido de /etc/naemon/conf.d	95
5.8	Contenido de /etc/naemon/conf.d/templates	95
5.9	Contenido del archivo de configuración del host	96
5.10	Contenido del archivo de configuración del services	96
5.11	Contenido del archivo de configuración del services	97
5.12	Información del host en Naemon	100
5.13	Información del host con cada servicio vinculado en Naemon	104
5.14	Información del estado del plugin de comprobación del puerto 80 sobre la página del artículo Hola, mundo de WordPress	114
5.15	Información del estado del plugin de comprobación de archivos	118
6.1	PNP4Nagios	123
6.2	Estructura de PNP4Nagios	128
6.3	HTTP en interfaz PNP4Nagios	130
6.4	PING en interfaz PNP4Nagios	130
6.5	Gráfica del servicio HTTP	131
6.6	Gráfica del servicio PING	131

Capítulo 1

Introducción

El objetivo de este proyecto es el completo despliegue de la herramienta **Naemon** a partir de contenedores **Docker**, apoyándonos de la creación de un sistema el cual analizaremos más adelante.

1.1. Objetivos

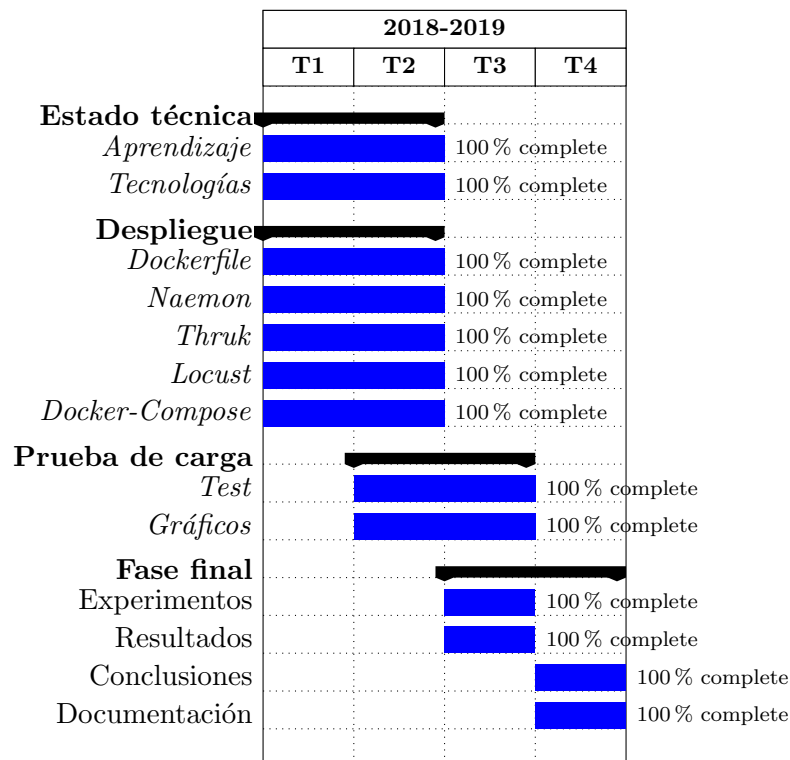
Para poder realizar el despliegue se va estructurar el proyecto de la siguiente forma mediante una serie de objetivos:

- Revisión bibliográfica del estado del arte. Este objetivo pretende explorar el concepto de monitorización sobre cualquier punto objetivo, como puede ser una red o una aplicación, en nuestro caso llegaremos a la realización del modelado de un sistema como puede ser WordPress, para poder adquirir el conocimiento para plantearnos entre varias herramientas a utilizar, llegar a la elegida por este presente proyecto, la herramienta conocida como Naemon.
- Profundizaremos el conocimiento de Naemon a través de su despliegue utilizando el conjunto de tecnologías conocidas como contenedores, aprovechando para introducir así este concepto realizando una comparación con otros servicios.
- Se creará un sistema de cargas sintéticas, utilizando herramientas de análisis y mediciones del desempeño de varios servicios establecidos en este proyecto, para así obtener el propósito de crear una concurrencia real, llegando a un sistema final de comprobación.

- Realización de representación de la carga con la recopilación de datos obtenidos mediante complementos ofrecidos por la herramienta Naemon.

1.2. Planificación del proyecto

- **T1:** Primer trimestre. Septiembre - Diciembre 2018
- **T2:** Segundo trimestre. Enero - Marzo 2019
- **T3:** Tercer trimestre. Abril - Junio 2019
- **T4:** Cuarto trimestre. Junio - Septiembre 2019



Capítulo 2

Estado del arte

A lo largo de este capítulo y a la vez del documento, se expondrá de forma sencilla y concisa, el concepto de monitorización, por el cual podemos encontrar los diferentes tipos de herramientas que nos ayudarán a realizar una monitorización exhaustiva del elemento que queremos examinar con exactitud, esto lo iremos elaborando y trabajando en este Trabajo Fin de Grado (TFG).

Veremos las herramientas actuales en el mercado y tras seleccionar la más apropiada para nuestro caso, entraremos en detalle del funcionamiento, el despliegue de la herramientas y las pruebas a realizar, todo esto irá explicado en los próximos capítulos.

2.1. Monitorización

A la hora de hablar sobre el concepto de **monitorización** de sistemas, tenemos que introducirnos en un concepto que nace con el objetivo de realizar un control exhaustivo sobre la red y los dispositivos que la integran de tal forma que se puedan gestionar las incidencias que vayan ocurriendo con el tiempo de una forma más rápida y eficaz.

Podemos encontrarnos ante el concepto de **monitorización proactiva**, que se trata de aquella en la que se toman las medidas preventivas y consecuentes con la información que se obtienen de todos los dispositivos que se encuentran conectados a una red, para evitar posibles eventos que hacen que se interrumpan el correcto funcionamiento de alguno de ellos.

Para poder abordar este concepto se utilizan protocolos que permitan la comunicación de la red con la herramienta.

Como estaremos analizando sistemas o cualquier dispositivo donde toda la información relevante será enviada a través de la red, será importante aplicar el uso del protocolo **SNMP** [1].

SNMP, sus siglas vienen de la palabra en inglés Simple Network Management Protocol, este protocolo pertenece a la capa de aplicación del modelo ISO, y como hemos mencionado permite el intercambio de información amplia sobre los diferentes dispositivos de red.

Este protocolo tiene dos formas de funcionar, la primera forma recibe el nombre de **polling** y la segunda forma recibe el nombre de **traps**.

El **polling** consiste en lanzar consultas de forma remota además de forma activa o demanda, realizando una operación síncrona de consulta. Los **traps** son mensajes que envían los dispositivos SNMP a una dirección configurada basándose en cambios o eventos, de forma asíncrona.[2]

2.1.1. Ventajas y desventajas de SNMP

La **principal ventaja** de **SNMP** es que es un estándar abierto. Los protocolos abiertos están diseñados para combatir el esfuerzo y los costos desperdiciados cuando un fabricante desarrolla su propio protocolo "patentado" que solo soportará.

Las **desventajas de SNMP** son tales que al poder obtener cualquier dato como una IP (ya sea una IP fuente o una IP destino), una MAC, un puerto o un protocolo, hacen que su utilidad se encuentre limitada a lo anterior descrito, ya que no facilita un procesamiento del tráfico, ni una obtención de datos más complejos sobre cualquier red. Por lo que la visibilidad se encontraría considerablemente reducida a las posibilidades de las consultas.

Se encuentra vinculada con cualquier estándar de comunicaciones poco utilizado. Aún así, el lanzamiento de SNMPv3 agregó nuevas opciones de cifrado y privacidad que nunca antes habían existido dentro de SNMP.

SNMP es un protocolo bastante detallado. Los mensajes detallados se envían entre dispositivos, no solo pequeños códigos preestablecidos. Este inconveniente se ha vuelto bastante pequeño en la mayoría de las aplicaciones, ya que el ancho de banda se ha disparado en los últimos años.

Actualmente, existen múltiples herramientas capaces de realizar la tarea de monitorización a través de aplicaciones o software que se encargan de la monitorización de cualquier red, dispositivo o sistema.

En el punto siguiente llevaremos a cabo una comparativa entre las posibles herramientas que podemos encontrar, basándonos en la documentación ofrecida por sus páginas oficiales de producto.

Existen una gran cantidad de herramientas, en este documento solo vamos a explicar las que se encuentran al alcance de cualquiera, por lo que nos será de preferencia comparar herramientas **OpenSource**.

Una herramienta **OpenSource** destaca por el hecho de que cuenta con la posibilidad de compartirse, modificarse y estudiarse su código abierto con la licencia que permite al usuario acceder a él con total libertad.

Realizaremos una enumeración de las más importantes, añadiendo además sus ventajas y desventajas de cada una y realizando la correspondiente **comparativa**.

2.2. Comparación herramientas de monitorización OpenSource

Las **herramientas de monitorización** se caracterizan unas de otras, por la manera en que realizan el intercambio de información entre los dispositivos. Por ello, en este apartado se destacará de las principales herramientas, los aspectos que las hacen destacar frente a otras y las carencias que estas presentan.

Estas aplicaciones deberían poder darnos la facilidad de encontrar la información acerca de cualquier dispositivo incluido dentro de la red o cualquiera de los componentes en tiempo real.

Entre estos componentes podemos encontrar servidores físicos o virtuales, servicios de correo, routers, switches, medición de protocolos, etc.

Hay que tener en cuenta que las redes y nuevas tecnologías van avanzando de forma exponencial, por lo que estas herramientas también deben ser capaces de adaptarse a este proceso de evolución.

22 2.2.2. Comparación herramientas de monitorización OpenSource

La mayoría permitirán mostrarlo de forma gráfica con su propio entorno o con la ayuda de herramientas que cuentan con **interfaz GUI**, dicho término corresponde a la interfaz gráfica de usuario que es un tipo de interfaz de usuario que utiliza imágenes, iconos y menús para mostrar las acciones disponibles entre las que el usuario puede escoger en un dispositivo.

Su función es proporcionar un entorno visual amigable y sencillo de usar que facilite la comunicación del usuario con el equipo.

Una de sus utilidades es que nos ayudará a mostrar todos los mecanismos necesarios y visuales para nosotros y así mostrarnos con mayor facilidad cualquier problema.

2.2.1. Zabbix



Figura 2.1: Zabbix

Zabbix [3] surge en 2001. Se trata de desarrollo completo, y su principal característica es que tiene una visión más holística de la monitorización, cubriendo rendimiento, no solo estados, ya que esta es una de las carencias más significativas de algunas herramientas de monitorización. Además de disponer de un sistema de gestión WEB que permite gestionarlo de forma centralizada, sin problemas en los ficheros de configuración.

Puede controlar aspectos como carga del dispositivo, actividad en la red, parámetros del sistema operativo, etc. Además ofrece la posibilidad de generar informes y gráficas en su interfaz web desarrollada en **PHP y JavaScript**.

Todo su contenido, capaz de recopilar datos de servidores y aplicaciones, por medio de agentes instalados en las máquinas cliente, se encuentran desarrollados en C. Soporta casi todos los **sistemas de gestión de bases de datos (SGBD)** desde MySQL, PostgreSQL, SQLite, Oracle o IBM DB2.

Zabbix tiene una interfaz web de gestión y que ésta se halla centralizada a través de la base de datos.

Posee un sistema de alertas para envío de notificaciones a través de email o SMS, pero además añade el uso de **Extensible Messaging Presence Protocol(XMPP)**[4](antes llamado Jabber), encargado de soportar mensajería instantánea, basado en **XML**.

Posee la ventaja de **detectar nuevos elementos añadidos a la red y puede establecer una jerarquía entre ellos**. Sin embargo, la capacidad de esta herramienta viene limitada a un número total de 1.000 nodos y hace que no esté a la misma altura que otras herramientas de características similares.

Otra desventaja es que no puede crear informes en tiempo real, además de no garantizar la seguridad SSL [5].

2.2.2. Cacti



Figura 2.2: Cacti

24 2.2.2. Comparación herramientas de monitorización OpenSource

Cacti [6] recopila información de los sistemas remotos para controlarlos. Además, presenta una importante gestión de las gráficas con los datos almacenados desde el momento en que empezó a monitorizar el sistema o servicio.

Principalmente, la aplicación está enfocada a la representación de los datos en gráficos para hacer que el usuario tenga mayor conocimiento de la gravedad de las diferentes alertas que aparezcan en la aplicación.

Esta herramienta es compatible con la incorporación de plugins o extensiones y tiene una comunidad grande de desarrolladores que proporcionan multitud de scripts para el mantenimiento de dicha aplicación, pero su base de datos no está en formato MySQL, si no en una serie de ficheros de texto que hacen que las consultas tengan un alto grado de dificultad.

La **parte negativa** de usar este software es la no generación automática de informes, que hace que los datos de usabilidad y rendimiento solo sean visibles instalando distintas herramientas de complementos, pero no suficientes para obtener todos los datos de la aplicación.

No dispone de entorno servidor ni de protocolo SSH[7], lo que dificulta la posibilidad de usar determinados chequeos o comprobaciones necesarias en algún servidor.

2.2.3. Pandora FMS



Figura 2.3: Pandora FMS

Pandora FMS[8] se trata de un sistema de monitorización español, desarrollado por la empresa Ártica. Pandora FMS nace en 2004. Al igual que Zabbix, es un desarrollo que parte de cero. Está destinada para grandes instalaciones. Puede realizar la monitorización de cualquier tipo de sistema, dispositivo o servicio. Lo anterior puede realizarlo teniendo el sistema de forma remota instalado a través de un cliente o a través de protocolos. Necesita un SGBD que por defecto es MySQL, donde almacenará toda la información a procesar.

Posee un interfaz web desarrollado en PHP5, aunque cuenta con la desventaja que es necesario el uso de Flash para visualizar cualquier gráfica. Su configuración puede gestionarse desde su interfaz web, por lo que cualquier persona capacitada con muy pocos conocimientos informáticos puede gestionar su mantenimiento. Esto se debe a su modularidad e intuitividad, orientada a objetos, que puede monitorizar todo tipo de servicios y parámetros con diferentes sensores mediante agentes instalados en los servidores que prestan el servicio.

En cuanto a la forma de su plataforma se estructura de forma similar a las demás, aunque cuenta con gran cantidad de complementos o plugins para el acceso web.

Cualquier cambio en la configuración, no implica reiniciar todo el sistema.

Al almacenar todos los datos en la base de datos es muy fácil la generación de gráficas y estadísticas en su interfaz web. Pandora FMS no es un sistema de monitorización de elementos críticos ya que no trabaja en tiempo real, ni tiene la opción de analizar logs o eventos.

Pandora FMS ha desarrollado su propio protocolo de transferencia de información, Tentacle, para intercambiar información con los clientes en los sistemas remotos.

La creación y mantenimiento de plugins no es muy dinámica ya que se trata de una comunidad bastante reciente.

Aunque con la versión OpenSource se puede cubrir la mayoría de las necesidades de cualquier red informática, además, posee una versión bajo licencia.

26 2.2. Comparación herramientas de monitorización OpenSource

2.2.4. Nagios



Figura 2.4: Nagios

Nagios[9] surge en 1999, considerándose una herramienta de las más populares. Reúne las principales ventajas nombradas en el caso anterior, si bien, para **Nagios**, dicha herramienta necesita de personal que cuente con suficientes conocimientos en cuanto a dicha herramienta o a la estructuración de sus ficheros para poder gestionar cualquier instalación y configuración en el servidor, así como, los clientes necesarios en los dispositivos remotos.

Desarrollada en **Perl**, integra un intérprete para los plugins desarrollados en este lenguaje, consiguiendo aumentar su rendimiento. Uno de sus puntos fuertes es la gran flexibilidad en su configuración, así como, la creación de nuevos plugins que se encarguen de monitorizar lo necesario para lo que todavía no existía ninguna opción o realizar alguna modificación a algún plugin existente para adaptarse a nuestras necesidades. Éstos pueden ser programados en diversos lenguajes como Perl, C, PHP, Python, etc. Existe una gran comunidad detrás de esta herramienta que la hace estar en continua evolución y actualizada.

Se pueden establecer jerarquías entre los dispositivos, establecer períodos diferentes de monitorización según las necesidades existentes, diferentes plantillas para controlar un mismo recurso, según sea la naturaleza del dispositivo o servicio a controlar. Permite las notificaciones por correo electrónico o SMS, diferenciando los niveles de criticidad.

Como **desventaja**, habría que destacar su bajo nivel de gráficas de estado, probablemente debido a no soportar un motor de base de datos que almacene toda la información, pero esto puede ser una ventaja en ciertas ocasiones, ya que necesita menos recursos. Además, los cambios que realicemos en cualquier fichero de configuración, necesitarán un reinicio del servicio para que pueda ser aplicado, esto es por el uso de **CGI's** [10], definen la forma en que un servidor Web puede interactuar con programas externos que generen contenido, se trata de una tecnología rápida y sólida, a diferencia de las demás herramientas que no aplican esta tecnología. Su arquitectura se basa en un único proceso, que ejecuta todas las tareas programadas.

Hay una variante comercial de este producto, **Nagios XI**, basándose en el volumen de nodos a monitorizar.

2.2.5. Naemon



Figura 2.5: Naemon

Naemon [11] surge en 2014 como un fork de **Nagios**, se trata de un sistema de código abierto y una aplicación de monitorización de red. Observa los hosts y servicios que especifique, le alerta cuando las cosas van mal y le avisa cuando mejoran. Naemon se basa en **Nagios 4.0.2**.

Naemon es el término general de la "*Suite Naemon*" completa que consta de dos partes, *Naemon Core* y la *Interfaz de Monitorización de Thruk*. En general, nos referiremos a Naemon Suite como solo Naemon.

Al ser un *fork* de Nagios posee todas sus características, pero debemos destacar una serie de aspectos:

- En la configuración de Naemon hay una innovación pequeña pero útil: los módulos de Event Broker no solo pueden cargarse directamente a través del archivo de configuración principal `naemon.cfg`, esto sería a través de la palabra clave "**define module ...**", sino que también se pueden dividir en sus propios archivos de configuración con sus propios parámetros.
- La **distribución** de Naemon es mejor que en Nagios. Para varias distribuciones, los paquetes están disponibles para una fácil instalación. En Nagios, por otro lado, siempre se debe compilar desde el código fuente, si desea usar una versión diferente a la contenida en el administrador de paquetes de la distribución.

2.3. Comparativa de Naemon

Se ha decidido realizar una comparativa por niveles entre Naemon y las demás herramientas que se han seleccionado de similares características, ya que dependiendo de la aplicación, se puede valorar muchos criterios que tienen más o menos consideración, a la hora de conocer la complejidad, usabilidad, disponibilidad y operatividad de una aplicación software libre de monitorización.

El nivel más determinante posiblemente sea el de aplicación, ya que para poder comparar las aplicaciones, serán determinantes muchas características de los mismos para poder analizarlas.

2.3.1. A nivel sistema operativo

Herramienta	OpenSource	Soporta MySQL	Entorno
Zabbix	Sí	Sí	Web y server
Cacti	Sí	No	Solo web
Pandora	Sí	No	Solo web
Nagios	Sí	Sí	Web y server
Naemon	Sí	Sí	Web y server

Figura 2.6: Comparativa a nivel de sistema

Como se puede observar, Nagios y Naemon cuentan con mejores características a nivel sistema operativo, ya que se pueden configurar los entornos y los protocolos, que a diferencia de otras aplicaciones no disponen, por lo que se hace más manejable y útil la utilización de Nagios y Naemon.

2.3.2. A nivel aplicación

Herramienta	Gráficas	Informes	Monitorización distribuida
Zabbix	Sí	No	No
Cacti	Sí	Sí	Sí
Pandora	Sí	Sí	No
Nagios	Sí	Sí	Sí
Naemon	Sí	Sí	Sí

Figura 2.7: Comparativa a nivel de aplicación

La visualización mediante gráficas, permite el poder visualizar en tiempo real todas las alertas que van apareciendo, exposición de la información de los chequeos mediante gráficas, para poder visualizar en tiempo real las alertas que van a apareciendo en el sistema.

2.3.3. A nivel extensiones

Herramienta	Plugins	Eventos	Logs	Logs avanzados
Zabbix	Compleja	Si	Si	No
Cacti	Compleja	No	Si	No
Pandora	Fácil	Si	Si	Si
Nagios	Fácil	Si	Si	Si
Naemon	Fácil	Si	Si	Si

Figura 2.8: Comparativa a nivel de extensiones

Como conclusión de las posibles herramientas que dan solución a algunos de los problemas que se están valorando, hay que destacar que existen varias que dan solución a algunos de los problemas, pero no al mismo nivel al que se pretende llegar, que con Nagios o Naemon se puede fácilmente.

A la hora de realizar la selección de la herramienta entre las que hemos comentado anteriormente, cabe destacar que todas cumplen con las necesidades que pueda surgir en cualquier red a monitorizar. Por lo que para seleccionar una, se ha decidido tomar como referencia que **Naemon** se trata de un fork de Nagios, por lo que contará con todas las funcionalidades de éste, además de contar con un mejor rendimiento, un funcionamiento más estable y, sobre todo, una nueva interfaz gráfica mucho más ligera, moderna y rápida. Esta nueva interfaz de Naemon, llamada, **Thruk Monitoring Webinterface**, es mucho más rápida, moderna y práctica respecto a la que aporta la suite Nagios.

2.4. Introducción de Naemon

2.4.1. Funcionamiento Naemon

Como se ha mencionado anteriormente Naemon está basado en Nagios exactamente en la **versión 4.0.2**, el cual cumple su misma función el poder representar la monitorización del sistema, a través de los host y servicios que queramos especificar, teniendo información sobre los cambios mediante las alertas que ofrece este sistema, además de informar cuando la situación empeora o mejora.[11]

Naemon se compone de dos partes, **Naemon Core** y **la interfaz de monitorización de Thruk**, hablaremos de ella más adelante en este capítulo, aunque es posible aplicar otra interfaz de monitorización, en todo el despliegue del proyecto se aplicará el uso de esta interfaz.

Entre las características que incluye Naemon podemos encontrar:

- Monitorización de servicios de red como SMTP, POP3, HTTP, NNTP, PING, entre otros.
- Monitorización de los recursos del host como puede ser la carga del procesador, el uso del disco, etc.
- Diseño de archivos plugins que permiten a los usuarios desarrollar fácilmente sus propios controles de servicio.
- Controles de servicios paralelos.
- Thruk Monitoring Webinterface para editar la configuración y ver el estado actual de la red, el historial de problemas, los archivos de registro, los informes de sla, los paneles de control, los procesos empresariales, etc.
- Capacidad para definir la jerarquía de hosts de la red mediante hosts "principales", lo que permite la detección y distinción entre los hosts que están inactivos y los que no están disponibles.
- Avisos de contacto cuando se producen problemas con el servicio o el host y se resuelven.
- Capacidad para definir controladores de eventos que se ejecutarán durante el servicio o eventos de host para la resolución proactiva de problemas.
- Rotación automática de archivos de registro.
- Soporte para implementar hosts de monitoreo redundantes.

Para la monitorización de hosts y servicios, Naemon al igual que Nagios utiliza **plugins**. Se tratan de componentes externos a los que Naemon les pasa información sobre el cometido que debe realizar, es decir, lo que debe comprobarse y los límites críticos y de advertencia. Una vez sea transmitida dicha información, los plugins harán las respectivas comprobaciones y analizarán los resultados.

Dicho resultado del chequeo podrá tener cuatro resultados: **OK**, **WARNING**, **CRITICAL** y **UNKNOWN**, además de información detallada de los mismos. Así de esta manera Naemon además de encontrar o notificar problemas, previene a cualquier organización que lo implemente, para no tenerlos en un futuro.

Toda esta información se recoge en un único servidor Naemon y así se puede acceder a toda la información de la infraestructura desde una única máquina.

2.4.1.1. Archivo de configuración principal

El archivo de configuración principal contiene una serie de directivas que afectan el funcionamiento del daemon de **Naemon-core**. Este archivo de configuración es leído tanto por el demonio **Naemon** como por **Thruk**, anteriormente conocido como "*CGIs*". Thruk proporciona una forma sencilla de editar la configuración de Naemon en la interfaz web sin tener que usar el terminal.

El **archivo de configuración principal** es normalmente llamado **naemon.cfg** y se encuentra en la carpeta **/etc/naemon/** [12].

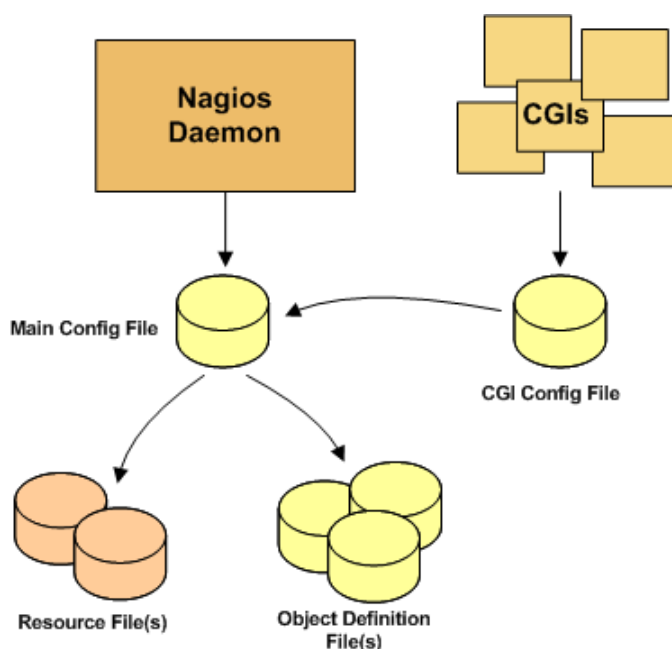


Figura 2.9: Configuración de Naemon[13]

La forma en la que se encuentran los **archivos de configuración de Naemon** es similar a como se encuentran en **Nagios**, al ser este un fork de él. Todo esto viene representado en la **figura 2.9**.

2.4.1.2. Objetos

La configuración de los distintos equipos y servicios a monitorizar se puede realizar a través de la carpeta **/etc/naemon/conf.d**, aquí podremos añadir equipos o hosts a la red, además podemos realizar una mejor gestión de la red.

Una de las características del formato de configuración de objetos de Naemon es que puede crear definiciones de objetos que heredan de otros.

A continuación, se explican los distintos objetos más interesantes que podemos añadir en esta **carpeta**:

2.4.1.2.1. Hosts Desde la carpeta **/etc/naemon/conf.d** se pueden añadir equipos a la configuración. Para ello, solo hay que crear un fichero **hosts.cfg** y además se debe conocer la dirección IP del dispositivo, el tipo de dispositivo (equipo Linux, Windows server, impresora, router o switch), sus ajustes preestablecidos, establecer los periodos de chequeo y de notificación.

A parte de esto se pueden definir relaciones entre hosts mediante el objeto **parent hosts** que permite definir la topología de la infraestructura. Los equipos padres suelen ser routers, switches, firewalls, etc. que se encuentran entre el equipo de monitorización y un host remoto y que se encargan de transmitir tráfico de paquetes entre ambos. Si el estado de una máquina es inaccesible puede deberse a que su **parent host**, es decir, su padre original, se haya caído. [12]

Además, se pueden crear grupos que engloben a varios dispositivos que por ejemplo vayan a utilizar los mismos servicios o tengan características similares para facilitar así su monitorización y organización. El host se guardará en una base de datos.

Uno de los principales beneficios es que esta carpeta permite a los usuarios definir plantillas y ajustes preestablecidos que pueden aplicarse al agregar hosts. Un host preestablecido contiene todos los servicios que se deben agregar a un host con todos los comandos vinculados y todos los valores predeterminados establecidos. También contiene el comando ***check-host-alive*** que se aplicará al nuevo host. Al añadir un host, el usuario debe ajustar algunos parámetros para adecuar la monitorización del host a su caso específico.

Naemon viene por defecto con unas plantillas predefinidas que hacen referencia a varios posibles tipos de máquinas. En cualquier caso, y como bien se ha dicho antes, el usuario puede añadir más sistemas desde ***/etc/naemon/conf.d*** en la parte de additional items y advanced items. Su formato de definición es el correspondiente a la **figura 2.10**.

define host {	
host_name	host_name
alias	alias
display_name	display_name
address	address
parents	host_names
hourly_value	#
hostgroups	hostgroup_names
check_command	command_name
initial_state	[o,d,u]
max_check_attempts	#
check_interval	#
retry_interval	#
active_checks_enabled	[0/1]
passive_checks_enabled	[0/1]
check_period	timeperiod_name
obsess_over_host	[0/1]
check_freshness	[0/1]
freshness_threshold	#
event_handler	command_name
event_handler_enabled	[0/1]
low_flap_threshold	#
high_flap_threshold	#
flap_detection_enabled	[0/1]
flap_detection_options	[o,d,u]
process_perf_data	[0/1]
retain_status_information	[0/1]
retain_nonstatus_information	[0/1]
contacts	contacts
contact_groups	contact_groups
notification_interval	#
first_notification_delay	#
notification_period	timeperiod_name
notification_options	[d,u,r,f,s]
notifications_enabled	[0/1]
stalking_options	[o,d,u]
notes	note_string
notes_url	url
action_url	url
icon_image	image_file
icon_image_alt	alt_string
vml_image	image_file
statusmap_image	image_file
2d_coords	x_coord,y_coord
3d_coords	x_coord,y_coord,z_coord
}	

Figura 2.10: Formato de definición de un objeto de tipo Host

A continuación se va explicar las distintas directivas que se pueden encontrar a la hora de definir un host:

- **host_name**: se usa para definir el identificador del host con un nombre corto.
- **alias**: se usa para definir un nombre o una descripción para identificar el host.
- **display_name**: se utiliza para definir un nombre alternativo que debe mostrarse en la interfaz web para este host.
- **address**: se utiliza para definir la dirección del host. Normalmente, esta es una dirección IP, aunque realmente podría ser cualquier cosa que desee.
- **parents**: se utiliza para definir una lista delimitada por comas de nombres cortos de los **hosts principales** para este host en particular.
- **hostgroups**: se utiliza para identificar los nombres cortos de los grupos de hosts a los que pertenece el host.
- **check_command**: se usa para especificar el nombre corto del comando que se debe usar para verificar si el host está activo o inactivo.
- **initial_state** [o,d,u]: por defecto, Naemon asumirá que todos los hosts están en estado UP cuando se inicia. Puede anular el estado inicial de un host utilizando esta directiva. *Las opciones válidas son: o = ARRIBA, d = ABAJO, y u = INACTIVO.*
- **max_check_attempts**: se utiliza para definir la cantidad de veces que Naemon volverá a intentar el comando de verificación de host si devuelve cualquier estado que no sea OK. Establecer este valor en 1 hará que Naemon genere una alerta sin volver a intentar la verificación del host.
- **check_interval**: se utiliza para definir el número de ‘unidades de tiempo’ entre las comprobaciones programadas regularmente del host.
- **retry_interval**: se utiliza para definir el número de ‘unidades de tiempo’ que debe esperar antes de programar una nueva verificación de los hosts.
- **active_checks_enabled**: se utiliza para determinar si las comprobaciones activas de este host están habilitadas.
- **passive_checks_enabled**: se utiliza para determinar si las comprobaciones pasivas están habilitadas o no para este host.

- **check_period:** se utiliza para especificar el nombre corto del período de tiempo durante el cual se pueden realizar verificaciones activas de este host.
- **process_perf_data:** se utiliza para determinar si el procesamiento de datos de rendimiento está habilitado o no para este host.
- **event_handler_enabled:** se utiliza para determinar si el controlador de eventos para este host está habilitado o no.
- **retain_status_information:** se utiliza para determinar si la información relacionada con el estado del host se retiene o no en los reinicios del programa.
- **retain_nonstatus_information:** se utiliza para determinar si la información que no es de estado sobre el host se retiene en los reinicios del programa. Esto solo es útil si ha habilitado la retención de estado utilizando la direc
- **contacts:** Esta es una lista de los nombres cortos de los contactos que se deben notificar siempre que haya problemas con este host.
- **contact_groups:** Esta es una lista de los nombres cortos de los grupos de contacto que se deben notificar siempre que haya problemas con este host.
- **notification_interval:** se utiliza para definir el número de “unidades de tiempo” que esperar antes de volver a notificar a un contacto que este servicio aún está inactivo o inaccesible.
- **notification_period:** se utiliza para especificar el nombre corto del período de tiempo durante el cual se pueden enviar notificaciones de eventos para este host a los contactos.
- **notification_options [d,u,r,f,s]:** se utiliza para determinar cuándo se deben enviar notificaciones para el host. Las opciones válidas son una combinación de uno o más de los siguientes: d = enviar notificaciones en estado ABAJO, u = enviar notificaciones en estado NO ALCANZABLE, r = enviar notificaciones en recuperaciones (estado correcto), f = enviar notificaciones cuando se inicia el host y deja de aletear, y s = envía notificaciones cuando comienza y termina el tiempo de inactividad programado.
- **notifications_enabled:** se utiliza para determinar si las notificaciones para este host están habilitadas o no.

2.4.1.2.2. Servicios El poder de esta herramienta de monitorización reside en los servicios que se pueden monitorizar. Estos pueden ser tanto servicios públicos como privados. [12]

Un **servicio** es “*público*” cuando es accesible a través de la red, ya sea en la red local o por Internet como **HTTP, POP3, IMAP, FTP y SSH**. Estos servicios y aplicaciones, así como sus protocolos, pueden ser monitorizados por Naemon sin necesidad de requerimientos de acceso especiales.

En cambio, los servicios “*privados*” no pueden ser monitorizados con Naemon sin la intervención de algún agente. Ejemplos de servicios privados asociados con los equipos son la carga de CPU, uso de memoria, uso en disco, cuentas de usuarios activos, información de procesos, etc. Estos servicios privados o atributos de los equipos usualmente no son expuestos a clientes externos. Esta situación requiere que un agente intermediario sea instalado en el equipo que se desea monitorizar esa información.

Sobre los dispositivos que hemos configurado anteriormente podemos añadir diferentes servicios de monitorización. Para ver la disponibilidad de dichas máquinas es conveniente añadir a todos ellos el servicio **PING** [14] que *nos hará saber si ese ordenador está configurado en la red y nos notificará si alguno de ellos cae en un momento dado, es decir, permite hacer una verificación del estado de una determinada conexión o host local*. **PING** se encuentra en la mayoría de sistemas operativos incluyendo Unix y Windows. Cabe mencionar que este servicio se encuentra desactivado en Windows por defecto.

A la hora de definir servicios es obligatorio fijar unos intervalos de chequeo del servicio y de notificación de los mismos por si alguno de ellos da algún problema a la hora de ser monitorizado. Los parámetros que definen estos periodos son los mismos que en los hosts.

En los servicios también se pueden definir plantillas con una serie de servicios a monitorizar definidos para una mayor facilidad, por ejemplo, si se quieren definir dichos servicios en múltiples equipos (en vez de tener que añadir cada servicio uno a uno, se añadirían todos los servicios de una vez).

Como en el caso de las máquinas, los servicios también se pueden agrupar. Estas agrupaciones sirven para manejar de una manera más efectiva los servicios, que sea más organizado a la hora de ver la información de los mismo en las interfaces web y para configurar dependencias entre equipos. Su formato de definición es el de la figura 2.11.

define service {	
<i>host_name</i>	<i>host_name</i>
hostgroup_name	hostgroup_name
<i>service_description</i>	<i>service_description</i>
display_name	display_name
parents	service_descriptions
hourly_value	#
servicegroups	servicegroup_names
is_volatile	[0/1]
<i>check_command</i>	<i>command_name</i>
initial_state	[o,w,u,c]
max_check_attempts	#
check_interval	#
retry_interval	#
active_checks_enabled	[0/1]
passive_checks_enabled	[0/1]
<i>check_period</i>	<i>timeperiod_name</i>
obsess_over_service obsess	[0/1]
check_freshness	[0/1]
freshness_threshold	#
event_handler	command_name
event_handler_enabled	[0/1]
low_flap_threshold	#
high_flap_threshold	#
flap_detection_enabled	[0/1]
flap_detection_options	[o,w,c,u]
process_perf_data	[0/1]
retain_status_information	[0/1]
retain_nonstatus_information	[0/1]
notification_interval	#
first_notification_delay	#
<i>notification_period</i>	<i>timeperiod_name</i>
notification_options	[w,u,c,r,f,s]
notifications_enabled	[0/1]
<i>contacts</i>	<i>contacts</i>
<i>contact_groups</i>	<i>contact_groups</i>
stalking_options	[o,w,u,c]
notes	note_string
notes_url	url
action_url	url
icon_image	image_file
icon_image_alt	alt_string
}	

Figura 2.11: Formato de definición de un objeto de tipo Servicios

A continuación se va explicar las distintas **directivas** que se pueden encontrar a la hora de definir un servicio, algunas son parecidas en cuanto a la funcionalidad como las usadas para definir un host:

- **host_name:** se utiliza para especificar los nombres abreviados de los hosts en los que el servicio se ejecuta o está asociado.
- **hostgroup_name:** se utiliza para especificar los nombres abreviados de los grupos de hosts en los que el servicio se ejecuta.^o con los que está asociado.
- **service_description:** se utiliza para definir la descripción del servicio, que puede contener espacios, guiones y dos puntos.
- **hourly_value:** se utiliza para representar el valor del servicio.
- **servicegroups:** se usa para identificar los nombres abreviados de los

grupos de servicios a los que pertenece el servicio. Varios grupos de servicio deben estar separados por comas.

- **is_volatile**: se utiliza para denotar si el servicio es volátil. Los servicios normalmente no son volátiles.
- **check_command**: se utiliza para especificar el nombre corto del comando que ejecutará Naemon para verificar el estado del servicio.
- **initial_state [o,w,u,c]**: por defecto, Naemon asumirá que todos los servicios están en estado OK cuando se inicia. Puede anular el estado inicial de un servicio utilizando esta directiva. Las opciones válidas son: o = OK, w = ADVERTENCIA, u = DESCONOCIDO, y c = CRÍTICO.
- **max_check_attempts**: se utiliza para definir la cantidad de veces que Naemon volverá a intentar el comando de verificación de servicio si devuelve cualquier estado que no sea OK.
- **check_interval**: se utiliza para definir el número de *unidades de tiempo* a esperar antes de programar la próxima verificación *regular* del servicio
- **retry_interval**: se utiliza para definir el número de *unidades de tiempo* a esperar antes de programar una nueva verificación del servicio.
- **active_checks_enabled**: se utiliza para determinar si las comprobaciones activas de este servicio están habilitadas o no.
- **passive_checks_enabled**: se usa para determinar si las comprobaciones pasivas de este servicio están habilitadas o no
- **check_period**: se utiliza para especificar el nombre corto del período de tiempo durante el cual se pueden realizar verificaciones activas de este servicio.
- **obsess_over_service|obsess**: determina si las comprobaciones para el servicio estarán *sobrecargadas* con el uso del comando *ocsp_command*.
- **check_freshness**: se utiliza para determinar si las comprobaciones de actualización están habilitadas o no para este servicio.
- **freshness_threshold**: se utiliza para especificar el umbral de actualización (en segundos) para este servicio.
- **event_handler**: se utiliza para especificar el nombre corto del comando que debe ejecutarse cada vez que se detecta un cambio en el estado del servicio.

- **event_handler_enabled**: se utiliza para determinar si el controlador de eventos para este servicio está habilitado o no.
- **low_flap_threshold**: se utiliza para especificar el umbral de cambio de estado bajo utilizado en la detección de aletas para este servicio.
- **high_flap_threshold**: se utiliza para especificar el umbral de cambio de estado alto utilizado en la detección de aletas para este servicio.
- **flap_detection_enabled**: se utiliza para determinar si la detección de flaps está habilitada o no para este servicio.
- **flap_detection_options [o,w,c,u]**: se usa para determinar qué estados de servicio utilizará la lógica de detección de aletas para este servicio. Las opciones válidas son una combinación de uno o más de los siguientes: o = estados OK, w = estados de ADVERTENCIA, c = estados CRÍTICOS, u = estados DESCONOCIDOS.
- **process_perf_data**: se utiliza para determinar si el procesamiento de datos de rendimiento está habilitado o no para este servicio.
- **retain_status_information**: se usa para determinar si la información relacionada con el estado del servicio se retiene o no en los reinicios del programa.
- **retain_nonstatus_information**: se usa para determinar si la información que no es de estado sobre el servicio se retiene o no en los reinicios del programa.
- **notification_interval**: se utiliza para definir el número de *unidades de tiempo* que esperar antes de volver a notificar a un contacto que este servicio todavía está en un estado no correcto.
- **first_notification_delay**: se usa para definir el número de *unidades de tiempo* que esperar antes de enviar la primera notificación de problema cuando este servicio entra en un estado no correcto.
- **notification_period**: se utiliza para especificar el nombre corto del período de tiempo durante el cual se pueden enviar notificaciones de eventos para este servicio a los contactos.
- **notification_options [w,u,c,r,f,s]**: se utiliza para determinar cuándo se deben enviar notificaciones para el servicio. Las opciones válidas son una combinación de uno o más de los siguientes: w = enviar notificaciones en estado de ADVERTENCIA, u = enviar notificaciones en estado DESCONOCIDO, c = enviar notificaciones en estado CRÍTICO, r = enviar notificaciones en recuperaciones (estado correcto), f = envía notificaciones cuando el servicio comienza y deja de aletear,

y `s` = envía notificaciones cuando comienza y termina el tiempo de inactividad programado.

- **notifications_enabled**: se usa para determinar si las notificaciones para este servicio están habilitadas o no.
- **contacts**: es una lista de los nombres cortos de los contactos que deben notificarse siempre que haya problemas con este servicio. Los contactos múltiples deben estar separados por comas.
- **contact_groups**: es una lista de los nombres cortos de los grupos de contacto que deben notificarse siempre que haya problemas con este servicio. Múltiples grupos de contacto deben estar separados por comas.

2.4.1.2.3. Comandos Los **comandos** definen cómo se deben hacer los chequeos de hosts y servicios, y cómo deben funcionar las notificaciones. Otra de sus funciones es restablecer el sistema automáticamente si es posible. [12]

La propia imagen de Naemon incluye una serie de comandos preestablecidos basados en plugins que son los que chequean los distintos servicios (SSH, HTTP, etc.) pero también el usuario puede añadir sus propios comandos.

Las definiciones de los comandos pueden contener macros, pero se debe asegurar de incluir solo aquellas que son "válidas" para las circunstancias en que se usará el comando.

A la hora de realizar la definición de un objeto de tipo **Comando** Naemon especifica una forma concreta a la hora de establecer sus directivas concretas, dicha forma viene representada en la figura 2.12 donde se puede apreciar el uso de la directiva **command_name** que se encarga de asignar un nombre al comando y **command_line** encargada de especificar el formato de línea del comando, es decir, cómo se verá en la **shell de Naemon**.

```
define command {
    command_name command_name
    command_line  command_line
}
```

Figura 2.12: Formato de definición de un objeto de tipo Comandos

2.4.1.2.4. Contactos Los contactos hacen referencia a diferentes personas que pueden ser los propietarios de diferentes máquinas, encargados de gestionar y solucionar problemas que pueden surgir en los dispositivos, destinatarios de notificaciones, etc.

A la hora de crear un contacto solo es obligatorio un nombre, un alias y una dirección de email. También se pueden definir los periodos de notificación de equipos y servicios y que tipo de notificaciones han de ser enviadas al contacto.

Los contactos se asignan a los usuarios que inician sesión en una de las interfaces web y todas las operaciones que se realicen a través de esa interfaz quedarán registradas como ese usuario y se permitirán dependiendo del acceso que tenga dicho usuario a determinados servicios.

Los contactos también pueden ser agrupados en **contact groups** para dividir tareas y responsabilidades entre, por ejemplo, los empleados de una empresa. Así, por ejemplo, las notificaciones de los servicios irán destinadas a un grupo de trabajo y las de los dispositivos a otro.

Su formato de definición es el de la figura 2.13.

define contact {	
contact_name	<i>contact_name</i>
alias	<i>alias</i>
contactgroups	<i>contactgroup_names</i>
minimum_value	#
host_notifications_enabled	[0/1]
service_notifications_enabled	[0/1]
host_notification_period	<i>timeperiod_name</i>
service_notification_period	<i>timeperiod_name</i>
host_notification_options	[d,u,r,f,s,n]
service_notification_options	[w,u,c,r,f,s,n]
host_notification_commands	<i>command_name</i>
service_notification_commands	<i>command_name</i>
email	<i>email_address</i>
pager	<i>pager_number or pager_email_gateway</i>
addressx	<i>additional_contact_address</i>
can_submit_commands	[0/1]
retain_status_information	[0/1]
retain_nonstatus_information	[0/1]
}	

Figura 2.13: Formato de definición de un objeto de tipo Contacto

2.4.1.2.5. Time Periods Los **time periods** son definiciones de periodos de tiempo en los que se pueden realizar los chequeos o se pueden enviar notificaciones de equipos y servicios. Estas definiciones se aplican a la hora de definir equipos y servicios en los parámetros `check period` y `notification period`.

Naemon trae predefinidos algún `time period` pero el usuario puede añadir los suyos propios. Solo hace falta un nombre único, un alias y el tiempo que se quiera definir.

Su formato de definición es el de la figura 2.14.

define timeperiod {	
<i>timeperiod_name</i>	<i>timeperiod_name</i>
<i>alias</i>	<i>alias</i>
[weekday]	<i>timeranges</i>
[exception]	<i>timeranges</i>
exclude	[<i>timeperiod1,timeperiod2,...,timeperiodn</i>]
}	

Figura 2.14: Formato de definición de un objeto de tipo Time-Period

2.4.1.2.6. Plantillas Como ya se ha mencionado anteriormente, Naemon permite crear **plantillas** para definir parámetros comunes que puedan ser aplicados a la hora de definir nuevos equipos y servicios.

Cuando se aplica una plantilla a un `host` o `servicio`, éste adquiere todas las propiedades definidas en la plantilla. Es posible aplicar varias plantillas a un único equipo. Si las plantillas especifican un mismo parámetro, predominará el valor del parámetro de la primera plantilla.

2.4.1.3. Thruk

Thruk[15] se trata de una interfaz de monitorización **multibackend**, es decir, puede emplear una gran variedad de `backends`, es decir, exactamente servidores `backend` como es el caso de Naemon, además de herramientas como Nagios y otras muchas más, todas ellas utilizando como API la llamada **LiveStatus API** [16]. Está diseñado para ser un reemplazo de la herramienta y básicamente cubrir las tareas de la herramienta al completo, añadiendo grandes instalaciones y haciendo así que incrementa su usabilidad.

Thruk viene organizado con la siguiente estructura reflejada en la figura 2.15. Teniéndose que su autenticación HTTP viene bajo el servidor web de Apache y corriendo bajo un servidor **Fast CGI** se conecta a Naemon a través de una conexión por el puerto correspondiente a TCP/IP.

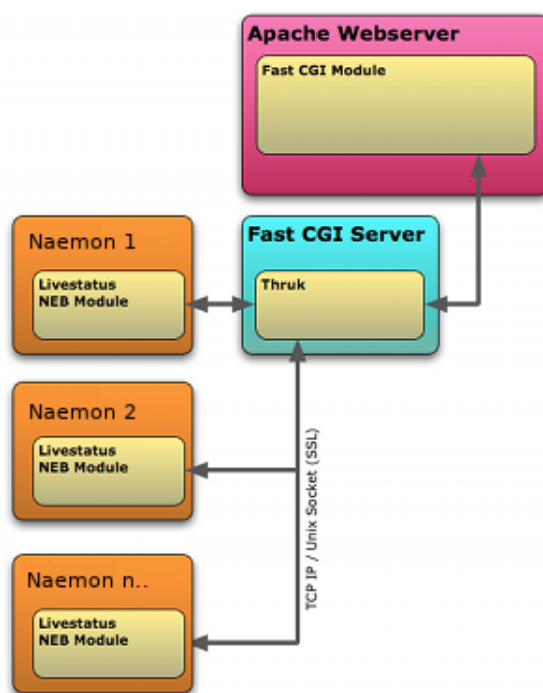


Figura 2.15: Estructura de conexión de Thruk

A través de la página <https://www.thruk.org/demo.html> se puede realizar pruebas de la demo de dicha interfaz.

Entre las principales **ventajas** de Thruk encontramos:

- Múltiples backends, esto es que podemos aplicar como servidor, se pueden incluir los siguientes: **Naemon, Nagios, Icinga y Shinken**. Supongamos que tiene una antigua instalación de Nagios heredada aún en producción, además de un nuevo servidor Naemon o Icinga 2. Con Thruk puede configurar ambos como backends para Thruk, y le dará una vista unificada para todos los hosts y servicios configurados. Una pestaña menos en su navegador.

- Thruk tiene muchas características excelentes, pero lo primero que notas es la velocidad. Incluso una configuración con miles de hosts y decenas de miles de servicios tiene tiempos de respuesta inferiores al segundo para los listados de servicios y los resultados de búsqueda.
- Thruk también tiene la opción de modificar su interfaz web clásica que lo hace un poco más fácil de usar y expande la funcionalidad de monitorización e informes.
- Muestra en vivo la información.
- Fácil incorporación de plugins, tanto plugins propios de Thruk, como plugins adicionales de Naemon en este caso.

Thruk viene de forma nativa con Naemon, y es una interfaz web de reemplazo completa y gratuita de código abierto para Nagios, Icinga y Shinken. Estas son herramientas flexibles para alertarnos cuando algo sale mal, y Thruk se encarga de garantizar una correcta monitorización.

Al ser una interfaz nativa de **Naemon** es preferible su utilización, puesto que su forma de integración y modificación de sus diferentes elementos será mucho menos complejo que con otras interfaces.

2.4.1.4. Plugins

A diferencia de muchas otras herramientas de monitorización, Naemon no incluye ningún mecanismo interno para verificar el estado de los hosts y servicios en su red. En su lugar, Naemon se basa en programas externos llamados **plugin** para hacer todo el trabajo.

Entonces los plugins son ejecutables o scripts (normalmente en perl, python, shell, entre otros) que se pueden ejecutar desde una línea de comandos para verificar el estado de un host o de un servicio concreto del host. Naemon usa los resultados de los plugins para determinar el estado actual de los hosts y servicios en su red. [17]

Naemon ejecutará un plugin siempre que sea necesario verificar el estado de un host o un servicio concreto del host. El plugin hace algo para realizar la comprobación y luego simplemente devuelve los resultados a Naemon. Naemon procesará los resultados que recibe del plugin y tomará las medidas necesarias.

Los **plugins** 2.16 actúan como una capa de abstracción entre la lógica de supervisión presente en el demonio de Naemon y los servicios y hosts reales que se están supervisando.

La ventaja de este tipo de arquitectura de plugins es que puede monitorear cualquier elemento.

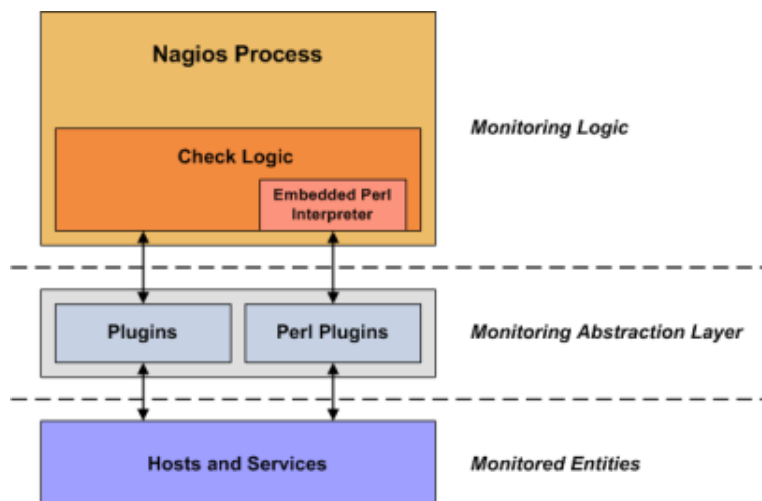


Figura 2.16: Funcionamiento de los plugins en Naemon

La **desventaja** de este tipo de arquitectura de plugin es el hecho de que Naemon no tiene reflejo de qué es lo que está monitorizando, simplemente rastrea los cambios en el estado de esos recursos.

Solo los plugin saben exactamente lo que están monitorizando y cómo realizar las comprobaciones reales.

Los **plugins de Naemon** no se distribuyen, pero se pueden descargar plugins de Naemon y plugins adicionales desde las siguientes **páginas**:

- **Proyecto de Monitorización de Complementos:** <http://monitoring-plugins.org>
- **Proyecto de complementos de Nagios:** <https://nagios-plugins.org/>
- **Nagios Exchange:** <http://exchange.nagios.org/>

Hemos mencionado que Naemon se trata de un fork de Nagios, por lo que la compatibilidad y uso de sus plugins será totalmente aceptable, por lo que podemos usar cualquier plugins de la base de datos de Nagios en Naemon.

Capítulo 3

Despliegue de Naemon

En este capítulo profundizaremos en la realización del **despliegue de Naemon**, introduciendo el entorno que se ha utilizado para su despliegue, comparando herramientas y seleccionando la que mejor se adecua a nuestras necesidades. Además se expondrá la configuración correspondiente para poder realizar ese despliegue desde la instalación rudimentaria hasta su prueba básica aplicando un plugin correspondiente.

3.1. Entorno del despliegue

A la hora de hablar de un **entorno de despliegue** nos referiremos a una instancia específica de una configuración de hardware y software establecida con el fin de instalar y ejecutar el software desarrollado para este uso.

Por lo que básicamente un entorno de despliegue lo trataremos como una colección de clústeres y servidores configurados que colaboran para proporcionar un entorno que aloje módulos de software concretos.

3.1.1. Alternativas a utilizar

A continuación se expondrán los posibles entornos a aplicar y se concluirá con la opción seleccionada.

3.1.1.1. Entorno nativo

Un **entorno nativo** lo definiremos como el entorno que se encuentra instalado directamente sobre un sistema anfitrión ejecutándose en hardware físico.

Al trabajar sobre el propio hardware contará con la ventaja de dedicar sus propios recursos a la hora de garantizar el mejor rendimiento.

Pero tiene la desventaja de que si existe algún problema de seguridad o alguna intrusión se efectuará sobre el propio sistema anfitrión. Otra desventaja es que es difícil controlar un backup en el caso de perder o no realizar ese backup.

Esta opción la descartaremos ya que es poco aconsejable utilizar entornos de desarrollo en nuestra máquina de forma local, ya que supone tener instaladas versiones del software que el desarrollador necesite en cada momento. Además sería una opción bastante compleja de controlar, teniéndose así una carga de trabajo adicional para el usuario, debido al tiempo que debe invertir en solucionar los problemas que puedan surgir debido a que algún empleado haya cambiado algo en su máquina local.

Por lo que será preferible que a la hora de desarrollar se trabaje bajo virtualización, evitando así los problemas que puedan surgir en una máquina local, ya que permite la creación de máquinas virtuales iguales para cada miembro de por ejemplo un equipo de desarrollo.

3.1.1.2. Entorno virtual o máquina virtual

El concepto de este término es que se trata de un sistema operativo completo funcionando de manera aislada sobre otro sistema operativo completo.

Este tipo de entorno permite compartir el hardware de forma que lo puedan utilizar varios sistemas operativos al mismo tiempo.

Un esquema de esta arquitectura es el mostrado en la figura 3.1.

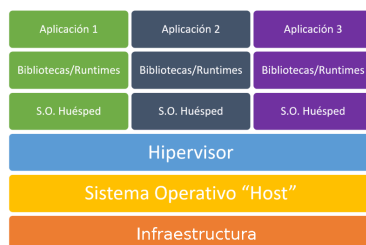


Figura 3.1: Arquitectura de un entorno virtual

Por debajo del sistema operativo siempre estará la infraestructura, por ejemplo el ordenador personal para realizar cualquier desarrollo, o si es por ejemplo un despliegue real se usará un servidor.

Para que una **máquina virtual** pueda ejecutarse es necesario la instalación del **hipervisor** que se trata de un software especializado en exponer los recursos hardware que están debajo del sistema operativo, de forma que puedan ser utilizados por otros sistemas operativos. Los hipervisores vienen como productos como Hyper-V, VirtualBox o VMWare, entre otros.

Gracias a todo esto podemos tener diferentes sistemas operativos ejecutándose en paralelo sobre la misma máquina física, cada uno con su memoria y espacio en disco reservados, y completamente aislados unos de otros.

3.1.1.3. Aplicación de contenedores

La filosofía de los contenedores es totalmente diferente a la de las máquinas virtuales. Si bien tratan también de aislar a las aplicaciones y de generar un entorno replicable y estable para que funcionen, en lugar de albergar un sistema operativo completo lo que hacen es compartir los recursos del propio sistema operativo "host" sobre el que se ejecutan.

Su esquema viene estructurado de la forma en la que viene representado en la figura 3.2.

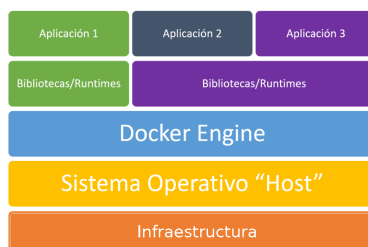


Figura 3.2: Estructura de un entorno con contenedores

En primer lugar debemos tener en cuenta que, en el caso de los contenedores, el hecho de que no necesiten un sistema operativo completo sino que reutilicen el subyacente reduce mucho la carga que debe soportar la máquina física, el espacio de almacenamiento utilizado y el tiempo necesario para lanzar las aplicaciones. Por lo tanto los contenedores son mucho más ligeros que las máquinas virtuales.

Cuando definimos una máquina virtual debemos indicar de antemano cuántos recursos físicos le debemos dedicar. En el caso de los contenedores esto no es así. De hecho no indicamos qué recursos vamos a necesitar, sino que en función de las necesidades de cada momento, el encargado de asignar lo que sea necesario para que los contenedores funcionen adecuadamente.

Esto hace que los entornos de ejecución en contenedores sean mucho más ligeros, y que se aproveche mucho mejor el hardware, además de permitir levantar muchos más contenedores que máquinas virtuales en la misma máquina física. Mientras que una máquina virtual puede tardar un minuto o más en arrancar y tener disponible nuestra aplicación, un contenedor se levanta y responde en unos pocos segundos. El espacio ocupado en disco es muy inferior con los contenedores al no necesitar que instalemos el sistema operativo completo.

Además para realizar despliegues avanzados de aplicaciones en contenedores hay que ir más allá y utilizar tecnologías que nos permiten orquestar y controlar los despliegues con muchas partes en ejecución.

Como desventajas se tiene que no pueden sustituir por completo la virtualización tradicional y no todo el mundo cuenta de un modelo de negocio que sea compatible a estos.

También aumentan riesgos de brechas de seguridad, además cuentan con menos flexibilidad a nivel de sistema operativo del contenedor.

Dejar de usar sistemas operativos separados implica una mejora en el rendimiento de la virtualización con contenedores, pero significa también tener un nivel más bajo de seguridad. En una virtualización del sistema operativo tienen efecto sobre todos los contenedores.

Todas las soluciones de **virtualización de contenedores Linux** actuales basan su funcionamiento en una serie de librerías o funcionalidades básicas del kernel y vertebran la arquitectura que da soporte a su funcionamiento. Estos componentes son:

- **CGROUPs**: se trata de una característica del kernel que permite limitar los recursos del sistema, es decir, limita, contabiliza y aísla el uso de los recursos como puede ser la CPU, la memoria, el disco, etc, para un conjunto de procesos.
- **NAMESPACES**: es un espacio donde uno o mas identificadores de proceso pueden co-existir. Esto permite que tengamos varios PID's agrupados, pero separados de otros sin que puedan ver qué recursos están utilizando cada uno de los espacios de nombres adyacentes. Existen 6 tipos básicos de **namespaces** relacionados con distintos aspectos del sistema:
 - **NETWORK namespace**: Aislamiento de red. Así, cada namespace de red tendrá sus propias interfaces, direcciones y puertos de red, tablas de enrutamiento, etc.
 - **PID namespace**: Aísla el espacio de identificadores de proceso. Un contenedor tendrá su propia jerarquía de procesos y su proceso padre o init (PID 1).
 - **UTS namespace**: Aísla el dominio y hostname, permitiendo a un contenedor poseer su propio dominio de nombres.
 - **MOUNT namespace**: Aísla los puntos de montaje de los sistemas de ficheros que puede ver un grupo de procesos. Este namespace fue el punto de partida, con chroot.
 - **USER namespace**: Aísla identificadores de usuarios y grupos. Así dentro un contenedor es posible tener un usuario con ID 0 (root) que se corresponda con un ID de usuario cualquiera en el host.
 - **IPC namespace**: Aísla la intercomunicación entre procesos dentro del espacio

En este trabajo usaremos los contenedores por las ventajas mencionadas. A continuación explicaremos algunas herramientas que permiten automatizar aplicaciones a través de contenedores:

3.1.1.3.1. Docker Engine **Docker** [18] [19] se trata de un proyecto de software libre que permite automatizar el despliegue de aplicaciones dentro de contenedores. Permitiendo empaquetar una aplicación en una unidad estandarizada para el desarrollo de software, que contiene todo lo necesario para su funcionamiento.



Figura 3.3: Docker

Usa recursos del sistema de forma totalmente aislada, permitiendo asignar recursos o combinaciones de estos recursos entre los procesos que se ejecutan en un sistema.

Los contenedores de imagen de **Docker** se pueden ejecutar de forma nativa en Linux y Windows. Sin embargo, las imágenes de Windows solo pueden ejecutarse en hosts de Windows y las imágenes de Linux pueden ejecutarse en hosts de Linux y hosts de Windows, donde host significa un servidor o una máquina virtual.

Permite centrarse en el código sin preocuparse de si funcionará en la máquina en la que se ejecutará. Esto acelera el proceso de mantenimiento y desarrollo de cada proyecto y, de cara a DevOps, permite que tanto los desarrolladores como los administradores de sistema pueden probar aplicaciones en un entorno seguro y exactamente igual en todos los casos.

A nivel de arquitectura y componentes, **Docker** funciona de forma que para proporcionar las características básicas de aislamiento y virtualización emplea las tecnologías que proporciona el Kernel de Linux.

Es un sistema que permite una enorme **portabilidad** ya que, debido al enorme auge que ha experimentado, la mayor parte de las grandes empresas comerciales de tecnologías de infraestructuras orientadas al cloud computing permiten ejecutar contenedores Docker de forma nativa desde sus plataformas cloud. Destacamos el caso de **Google Cloud**, **Amazon AWS** y **Microsoft Azure** que tienen plataformas nativas especializadas en la ejecución de contenedores Docker con múltiples funcionalidades y tipos de escalabilidad y alta disponibilidad.

Auspiciado por el éxito de las soluciones **Docker**, podemos encontrar en el mercado un gran numero de aplicaciones comerciales en formato contenedor Docker para ser usadas y arrancadas sin necesidad de conocimientos previos. Del mismo modo, podemos encontrar todo tipo de servicios que se distribuyen preparados para ser usados en formato contenedor Docker, lo que nos permite acceder a desplegar una enorme cantidad de aplicaciones y servicios sin prácticamente necesidad de un conocimiento previo de los pasos de su instalación, y lo que es más importante, con el soporte directo del fabricante de la solución.

Podemos ver a **Docker** además de como una herramienta para la creación de contenedores virtuales, como una capa más de abstracción y aislamiento que además aporta un sistema que facilita enormemente la gestión y administración de los contenedores.

3.1.1.3.2. OpenShift Software creado por **Red Hat**, fabricante detrás de una de las distribuciones Linux comerciales más extendidas, OpenShift es una solución que permite desplegar aplicaciones realizadas con cualquier stack de tecnologías.[20]



Figura 3.4: OpenShift

Básicamente, permite al desarrollador centrarse en su trabajo de creación de la aplicación y evitar preocuparse con el mantenimiento de los servicios o la escalabilidad de las aplicaciones. Es capaz de trabajar con lenguajes y frameworks diversos y permite modificar y desplegar las aplicaciones rápidamente bajo demanda, automatizando el proceso entero.

Cuenta con una interfaz bastante intuitiva y manejable. **Openshift** se basa en Docker y Kubernetes, pero añade una nueva capa.

Es ahora mismo una de las opciones más interesantes para **Cloud privadas**, ya que está basada en estándares abiertos, integra con las tecnologías open source de referencia y tiene un market en constante crecimiento.

Aunque está basada en código abierto, y cuenta con versiones gratuitas, no deja de ser un producto de **RedHat**. Por lo que contará con versiones de pago.

Como desventaja es que no deja instalar ciertas imágenes o utilizar el usuario root. Cuenta con gestión de imágenes de contenedores con **ImageStreams** Además tiene integración de la herramienta **Jenkins**.

3.1.1.3.3. CoreOS rkt (rocket) Es un motor de software de automatización de contenedores para ejecutar cargas de trabajo de aplicaciones de forma aislada de la infraestructura subyacente.[21]



Figura 3.5: CoreOS

Los contenedores CoreOS rkt operan en la mayoría de las principales distribuciones del sistema operativo Linux como un archivo binario que se integra con los sistemas y scripts de inicio de Linux, y opera de acuerdo con el modelo de proceso estándar de Unix, que es una relación padre-hijo.

Respecto a CoreOS Rkt, este sistema es similar a Docker ya que se encarga de la virtualización de aplicaciones en el propio sistema operativo principal.

Algunas de las principales características de este software es que también se ha diseñado pensando en la seguridad por defecto, incluye soporte para **SELinux** así como para ejecutar aplicaciones en máquinas virtuales aisladas. Otras características es que soporta varios sistemas de inicio como `systemd` y `upstart`, por último, también puede ejecutar imágenes Docker.

Como inconvenientes es que existen muchas menos integraciones de proveedores externos para **rkt** que para Docker.

Además **rkt** está optimizado para operar contenedores de aplicaciones, pero no soporta contenedores de sistemas operativos (full system container).

3.1.1.3.4. LXC (Linux Containers) A diferencia de Docker y CoreOs **rkt**, **LXC** es una tecnología de sistema que se encarga de generar contenedores de sistema.[22]

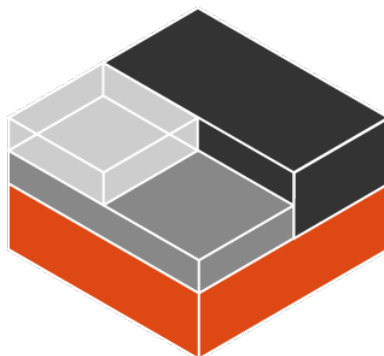


Figura 3.6: LXC

El que compartan el kernel otorga menos aislamiento al sistema principal y además le da acceso a los recursos de la máquina anfitriona, pero queda parcialmente limitado por espacios de procesos y de red.

Como **desventaja** podemos decir que no se utiliza de forma estándar para operar contenedores de aplicaciones. A excepción de Linux, no cuenta con ninguna implementación nativa para otros sistemas operativos. Esto es que también podemos ejecutarlo en otros sistemas operativos, como puede ser Mac o Windows, pero siempre como resultado de ejecutarlos embebidos dentro de un entorno Linux que se virtualiza de forma clásica sobre un anfitrión con otro sistema operativo.

Por este motivo, se considera que no es una ejecución nativa ya que es necesario virtualizar primero el sistema operativo Linux para poder, dentro de este, virtualizar contenedores.

3.1.2. Alternativa elegida: Uso de contenedores con Docker

Después de exponer todas las alternativas anteriores usaremos **Docker** por las siguientes razones:

- En cuanto al peso, es el más ligero ya que automatiza aplicaciones con muy poco espacio.
- Se puede desplegar cualquier contenedor en cualquier otro sistema, con lo que se ahorra el tener que instalar de nuevo los entornos.
- Es más seguro ya que utiliza por defecto la máxima confianza probable para que el software que contiene se ejecute en un ambiente totalmente controlado.
- Usa libcontainer que es un derivado de LXC, que se trata de una abstracción para admitir una gama más amplia de tecnologías de aislamiento.
- Permite una consola interactiva para el usuario.
- Es la opción más recomendada para montar entornos ad-hoc con tecnologías open source, ya sea en nubes públicas o privadas.

3.2. Desarrollo del despliegue de Naemon

En cuanto al software utilizado, el equipo cuenta con un sistema operativo **Pop!_OS 18.04** y al que se le ha realizado la instalación de **Docker**.

Pop!_OS se basa en Ubuntu, de la que toma sus dos grandes fundamentos: Linux 5.0 y GNOME 3.32. Cuenta con mejor soporte de hardware, mejor rendimiento y más opciones de configuración.

Por otro lado, el escritorio escapa de la disposición del de Ubuntu y se acerca al estilo de GNOME, complementándolo con su propio tema de aplicaciones e iconos, algunos de los cuales han sido rediseñados para esta versión siguiendo las líneas de diseño de GNOME, con diversas extensiones preinstaladas, para poder tener los iconos en el escritorio o para gestionar la batería.

Pop!_OS ofrece la posibilidad de reinstalar el sistema sin perder usuarios o datos. La distribución sigue las versiones de Ubuntu, pero no está enfocada en la reinstalación, sino en la actualización constante de una versión a otra con excepción de la LTS, para la que siguen manteniendo algunos de sus paquetes.

Además cuenta con dos formas de instalación o adaptación uno para **Intel/AMD** y otro para **Nvidia**.

El contenedor creado lanzará Naemon sobre una base Ubuntu Bionic (18.04).

La imagen base que utilizaremos será una versión minimalista de Ubuntu Bionic obtenida desde:
<https://github.com/phusion/baseimage-docker>.

Tienes Ubuntu instalado en Docker. Los archivos están ahí. Pero eso no significa que Ubuntu esté funcionando como debería.

Cuando se inicia su contenedor Docker, solo se ejecuta el comando CMD. Los únicos procesos que se ejecutarán dentro del contenedor es el comando CMD y todos los procesos que genera. Es por eso que todo tipo de servicios importantes del sistema no se ejecutan automáticamente: debe ejecutarlos usted mismo.

Su sistema de inicio, **Upstart**, asume que se está ejecutando en hardware real o virtualizado, pero no dentro de un contenedor Docker. Por eso utilizaremos esta imagen de **phusion**.

Esta imagen incluye la forma de iniciar el proceso de inicio de forma correcta, esto es que viene con un proceso de inicio **/sbin/my_init** que cosecha los procesos secundarios huérfanos correctamente y responde a **SIGTERM** correctamente. De esta manera, el contenedor no se llenará de procesos **zombies** y el comando **docker stop** funcionará correctamente.

Además soluciona **incompatibilidades APT con Docker**. Se encarga de ejecutar un demonio **syslog** para que los mensajes importantes del sistema no se pierdan. Otra tarea que incluye es que ejecuta un demonio **cron** para que los **cronjobs** funcionen. Cuenta con un **servidor SSH** que le permite iniciar sesión fácilmente en su contenedor para inspeccionar o administrar cosas. El **daemon SSH** está deshabilitado de forma predeterminada.

Cuenta con el demonio **runit** que es utilizado para la supervisión y gestión de servicios. Mucho más fácil de usar que **SysV init** [23] y admite reiniciar demonios cuando se bloquean. Mucho más fácil de usar y más liviano que **Upstart**.

Incluye una herramienta personalizada para ejecutar un comando como otro usuario. Es más fácil de usar que **su**, tiene un vector de ataque más pequeño que **sudo**, y a diferencia de **chpst** esta herramienta configura **\$HOME** de forma correcta. Disponible como **/sbin/setuser**.



Figura 3.7: Estructura del contenedor Docker con Naemon y Ubuntu de forma minimalista

3.3. Creación de Dockerfile

Para poder realizar el despliegue tendremos que crear el fichero **Dockerfile**. Con este fichero construiremos la imagen de Naemon de forma automática, leyendo las instrucciones que le indiquemos.

Se trata de un documento de texto que contiene todas las órdenes a las que un usuario dado puede llamar, desde la línea de comandos, para crear una imagen.

Los pasos principales para crear una imagen a partir de un fichero Dockerfile son:

- Crear un nuevo directorio que contenga el fichero, con el guión y otros ficheros que fuesen necesarios para crear la imagen.
- Crear el contenido.
- Construir la imagen mediante el comando `docker build`.

La sintaxis para el comando es:

```
1 $ docker build [opciones] RUTA | URL | -
```

Las opciones más comunes son:

- **-t**, nombre [:etiqueta]. Crea una imagen con el nombre y la etiqueta especificada a partir de las instrucciones indicadas en el fichero. Se recomienda usar este parámetro cuando se quiere asignar un nombre a la imagen.
- **-no-cache**. Por defecto, Docker guarda en memoria caché las acciones realizadas recientemente. Si se diese el caso de que ejecutamos un `docker build` varias veces, Docker comprobará si el fichero contiene las mismas instrucciones y, en caso afirmativo, no generará una nueva imagen. Para generar una nueva imagen omitiendo la memoria caché utilizaremos siempre esta opción.
- **-pull**. También por defecto. Docker solo descargará la imagen especificada en la expresión FROM si no existe. Para forzar que descargue la nueva versión de la imagen utilizaremos esta opción.

- **–quiet.** Por defecto, se muestra todo el proceso de creación, los comandos ejecutados y su salida. Utilizando esta opción solo mostrará el identificador de la imagen creada.

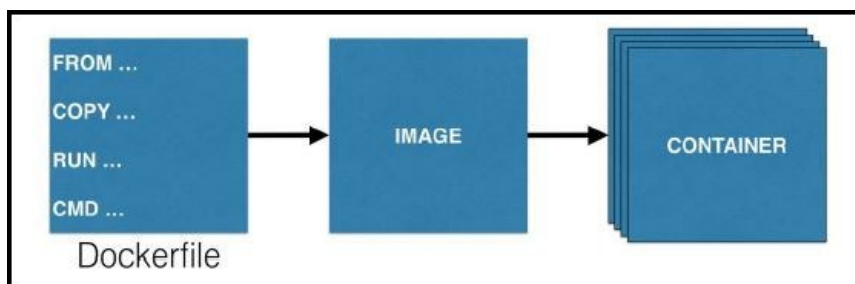


Figura 3.8: Estructura de un fichero Dockerfile

Para crear este fichero **Dockerfile** como el figurado en 3.8 antes tenemos que aprender los comandos disponibles:

- **MAINTAINER** : Nos permite configurar datos del autor, principalmente su nombre y su dirección de correo electrónico.
- **ENV** : Configura las variables de entorno.
- **ADD** : Esta instrucción se encarga de copiar los ficheros y directorios desde una ubicación especificada y los agrega al sistema de ficheros del contenedor. Si se trata de añadir un fichero comprimido, al ejecutarse el guión lo descomprimirá de manera automática.
- **COPY** : Es la expresión recomendada para copiar ficheros, similar a ADD.
- **EXPOSE** : Indica los puertos en los que va a escuchar el contenedor. Hay que tener en cuenta que esta opción no consigue que los puertos sean accesibles desde el host; para esto debemos utilizar la exposición de puertos mediante la opción -p de docker run, tal y como explicamos en un artículo anterior.
- **VOLUME** : Esta es una opción que muchos usuarios de la Web estaban esperando como agua de mayo. Nos permite utilizar en el contenedor una ubicación de nuestro host, y así, poder almacenar datos de manera permanente. Los volúmenes de los contenedores siempre son accesibles en el host anfitrión, en la ubicación: `/var/lib/docker/volumes/`

- **WORKDIR** : El directorio por defecto donde ejecutaremos las acciones.
- **USER** : Por defecto, todas las acciones son realizadas por el usuario root. Aquí podemos indicar un usuario diferente.
- **SHELL** : En los contenedores, el punto de entrada es el comando `/bins/sh -c` para ejecutar los comandos específicos en CMD, o los comandos especificados en línea de comandos para la acción run.
- **ARG** : Podemos añadir parámetros a nuestro Dockerfile para distintos propósitos.

A continuación indicaremos los pasos a seguir para instalar Naemon y la forma en que se ha indicado en dicho **Dockerfile**.

3.3.1. Instalación de imagen base phusion

La base de descarga será la siguiente <https://github.com/phusion/baseimage-docker>, donde podemos encontrar que dicha base es más eficiente que las bases comunes de Docker conocidas como **Alpine** y **BusyBox** en cuanto a la entrada y salida de sus datos de red.

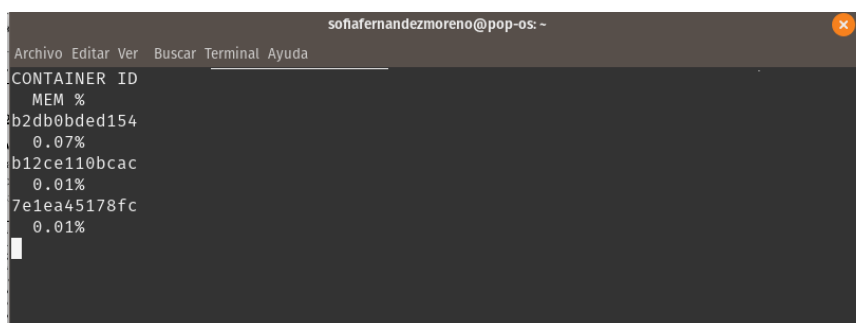


Figura 3.9: Comparativas de estadísticas

En la figura 3.9 se referencia al nombre **hungry_poitras** con la imagen de la base de **phusion**, el nombre de **vigorous_wing** con **Busybox** y **objective_crazy** con **Alpine**.

Por lo que tendremos que añadir en el Dockerfile las siguientes líneas:

```
1 FROM phusion/baseimage:0.11
2 MAINTAINER Sofia <chui274@gmail.com>
```

3.3.2. Instalación paquetes

Debemos instalar las dependencias y los paquetes para que Naemon se pueda ejecutar perfectamente. Para ello debemos instalar el entorno LAMP en donde vayamos a aplicar Naemon. El comando que introduciremos en nuestro Dockerfile será el siguiente:

```
1 RUN apt-get update && \
2 DEBIAN_FRONTEND=noninteractive \
3 apt-get install -y \
4 apache2\
5 apache2-utils\
6 libapache2-mod-fcgid\
7 libfontconfig1\
8 libjpeg62\
9 libgd3\
10 libxpm4\
11 xvfb\
12 ssmtp\
13 ruby\
14 python2.7\
15 python-boto\
16 perl\
17 libwww-perl\
18 libcrypt-ssleay-perl
```

Al principio le hemos indicado actualizar los repositorios por si existe alguna versión reciente en nuestro sistema.

Además hemos indicado para usar la sentencia siguiente:

```
1 DEBIAN_FRONTEND=noninteractive
```

para indicar que queremos instalar y establecer el entorno apt-get de forma no interactivo y usar el script de shell en Dockerfile.

3.3.3. Instalación clave GPG

El siguiente paso a seguir es agregar el repositorio de **Consol Labs** para crear la base de datos apt. Para ello debemos importar la clave GPG y así seguidamente generaremos las claves públicas para los servidores.

Por lo que habrá que añadir al documento **Dockerfile** las siguientes líneas para realizar lo anterior especificado:

```
1 RUN curl -s "https://labs.consol.de/repo/stable/RPM-GPG-KEY" | apt-key add -
2 RUN gpg --keyserver keys.gnupg.net --recv-keys F8C1CA08A57B9ED7
3 RUN gpg --armor --export F8C1CA08A57B9ED7 | apt-key add -
4 RUN echo "deb http://labs.consol.de/repo/stable/ubuntu $(lsb_release -cs) main" > /etc/apt/sources.list.d/labs-consol-stable.list
```

3.3.4. Instalación repositorio

Realizado la instalación de la clave GPG debemos actualizar los repositorios y una vez actualizados los repositorios instalaremos la aplicación Naemon, además de que instalaremos los plugins correspondientes de Nagios que son compatibles con este, para ello introduciremos los siguientes comandos al Dockerfile:

```
1
2 RUN apt-get update &&\
3 DEBIAN_FRONTEND=noninteractive apt-get install -y \
4 nagios-nrpe-plugin\
5 naemon=1.0.10
```

Le volvemos a indicar que queremos el entorno de forma no interactiva, además a la hora de realizar las instalaciones se le indicará, a través del **parámetro -y**, que confirmamos la instalación, ya que si no se confirma con anterioridad no realizará la creación de la imagen ya que no Docker no ofrece los permisos suficientes.

3.3.5. Inicialización de directorios

Una vez finalizada la instalación de Naemon, tendremos que indicar los directorios en los que vamos a trabajar, ya que el usuario desconoce los directorios internos de Naemon, para ello realizaremos la inicialización del directorio de datos creando unas variables de entorno.

Para ello crearemos un archivo llamado **data_dirs.env** donde se recorrerá la variable global **DATA_DIRS** que se compondrá de los directorios que queremos tener en nuestro contenedor, es decir, para poder tener acceso a los directorios con los que vamos a trabajar.

Dicho archivo **.env** permite personalizar las variables de entorno de trabajo individuales. Como hemos mencionado recorreremos la variable **DATA_DIRS** ingresando los directorios:

- /etc/naemon
- /var/log/naemon
- /etc/thruk
- /var/log/thruk
- /usr/lib/naemon/plugins

Introduciremos dentro de la variable para agregar también el directorio de instalación de Thruk.

La forma de ir introduciendo los directorios lo haremos a través de forma de lenguaje bash de la siguiente manera:

```
1 i=0
2 DATA_DIRS[((i++))]="/etc/naemon"
3 DATA_DIRS[((i++))]="/var/log/naemon"
4 DATA_DIRS[((i++))]="/etc/thruk"
5 DATA_DIRS[((i++))]="/var/log/thruk"
6 DATA_DIRS[((i++))]="/var/www"
7 DATA_DIRS[((i++))]="/usr/lib/naemon/plugins"
8 export DATA_DIRS
```

3.3.5.1. Creación script de inicialización: init.bash

Ahora una vez creada la variable de entorno con la que vamos a trabajar para poder operar con los directorios necesitamos inicializar el diseño del directorio de datos personalizado, para ello crearemos un script bash donde nos encargaremos de incluir en una variable **datadir** todos los datos recogidos de la variable de entorno y moverlos a la plantilla de cada directorio, seguidamente realizaremos un enlace simbólico entre los directorios creados y los directorios reales.

Es decir, a través del fichero **data_dirs.env** accederemos a la variable de entorno **DATA_DIRS** y por cada iteración del mismo moveremos dichas rutas a una plantilla y luego como hemos mencionado se creará el enlace simbólico.

El script es el siguiente:

```
1 #!/bin/bash
2
3 source /data_dirs.env
4
5 mkdir -p /data
6 for datadir in "${DATA_DIRS[@]"; do
7 mv ${datadir} ${datadir}-template
8 ln -s /data/${datadir#/*} ${datadir}
9 done
```

3.3.5.2. Copia de archivos

A través de Dockerfile y con el comando ADD copiaremos los archivos anteriores data_dirs.env e init.bash a sus repestivos, es decir, realizaremos lo siguiente:

```
1 ADD data_dirs.env /data_dirs.env
2 ADD init.bash /init.bash
```

Con esto queremos copiar el contenido de data_dirs.env en el nuevo archivo /data_dirs.env, al igual pasa con init.bash que copiaremos su contenido en la raíz del contenedor, es decir, en /init.bash.

Esto lo realizaremos para que el **Dockerfile pueda tener acceso a dichos datos y así poder operar**, ya que por defecto Docker no sabe de la existencia de dichos archivos sino los introducimos en el contenedor.

3.3.5.3. Asignación de permisos

Para poder ejecutar dicho script init.bash es necesario asignar una serie de permisos, para ello le indicaremos que queremos asignarle el permiso 755, es decir, este permiso se ocupa de que el propietario del fichero pueda leer, escribir y ejecutar el archivo. Todos los otros puedan leer y ejecutar el archivo. Este ajuste es común para los programas que son utilizados por todos los usuarios.

Además le indicaremos también a través del comando `sync` que fuerce la grabación de los datos de la caché y así solventar la presencia de errores y reducir el tamaño de procesamiento. Esto lo indicaremos en el fichero Dockerfile de la siguiente forma:

```
1 RUN chmod 755 /init.bash &&\
2 sync && /init.bash &&\
3 sync && rm /init.bash
```

Además le indicaremos que ejecute el script `init.bash` para poder inicializar la información de los directorios.

3.3.6. Cargar datos en carpeta raíz

Como se mencionó anteriormente se creaba el directorio **data**, entonces para poder trabajar con el contenedor estableceremos el punto de montaje de Naemon en dicho directorio eso lo realizaremos con el comando **VOLUME** de Docker, que básicamente creará un punto de montaje con el nombre especificado y lo marca como que contiene volúmenes montados externamente desde el host nativo u otros contenedores.

El valor puede ser una *matriz JSON*, por ejemplo, **VOLUME** `[“/var/log/”]` o una cadena simple con varios argumentos, como **VOLUME** `/var/logo`. Cuando realicemos el comando *run* de Docker, se inicializará el volumen recién creado con cualquier dato que exista en la ubicación especificada dentro de la imagen base. Por lo tanto esto lo realizaremos en el fichero Dockerfile de la siguiente manera:

```
1 VOLUME ["/data"]
```

3.3.7. Protocolo de salida

Para poder utilizar Naemon en un servidor web debemos realizar una conexión a través de los nodos y para ello utilizaremos el protocolo **TCP** desde el puerto 80 de HTTP.

Esto se podrá realizar a través de EXPOSE, qu su función es la de informar a Docker que el contenedor escucha en los puertos de red especificados en el tiempo de ejecución. Puede especificar si el puerto escucha en TCP o UDP, y el valor predeterminado es TCP si no se especifica el protocolo.

Dicha instrucción no publica realmente el puerto. Funciona como un tipo de documentación entre la persona que construye la imagen y la persona que ejecuta el contenedor, acerca de los puertos que deben publicarse.

Para publicar realmente el puerto cuando se ejecuta el contenedor, se debe usar el flag **-p** en el comando `docker run` para publicar y asignar uno o más puertos, o el flag **-P** para publicar todos los puertos expuestos y asignarlos a puertos de alto orden.

Por defecto, EXPOSE asume TCP. También puede especificar UDP. En nuestro Dockerfile usaremos la siguiente línea para comunicar con TCP.

```
1 EXPOSE 80/tcp
```

3.3.8. Creación de imagen y ejecución de Dockerfile

En esta sección especificaremos como podemos ejecutar a través de los propios comandos de Docker nuestra imagen, para ello realizaremos los siguientes pasos:

3.3.8.1. Creación de ENTRYPOINT

Primero utilizaremos el comando ENTRYPOINT en nuestro Dockerfile, ya que la funcionalidad de este es permitir configurar un contenedor que se ejecutará como un tipo de ejecutable. ENTRYPOINT puede tener dos formas:

- **ENTRYPOINT [ejecutable,parametro]** Formato ejecutable
- **ENTRYPOINT comando parametro** Formato shell

En nuestro caso se ha elegido la primera opción, ya que previamente hemos creado un fichero `bash` donde realizaremos las ejecuciones que contaremos a continuación. Previamente se ha creado en el Dockerfile las líneas para copiar dicho `bash` en la raíz del contenedor y darle la asignación de permisos de tipo 755 a dicho `bash`:

```
1 ADD run.bash /run.bash
2 RUN chmod 755 /run.bash
```

3.3.8.1.1. Fichero bash de ejecución: run.bash Anteriormente ejecutábamos la asignación de permisos de ejecución del fichero run.bash, para realizar la ejecución realizamos:

```
1 ENTRYPOINT ["/run.bash"]
```

Con esto ejecutará dicho fichero bash que vamos a comentar a continuación:

Primero realizaremos la configuración de Naemon de forma externa a través del volumen **/data**

```
1
2 source /data_dirs.env
3 DATA_PATH=/data
4
5 for datadir in "${DATA_DIRS[@]"; do
6     if [ ! -e "${DATA_PATH}/${datadir#/*}" ]
7     then
8         echo "Installing ${datadir}"
9         mkdir -p ${DATA_PATH}/${datadir#/*}
10        if [ "$(ls -A ${datadir}-template 2> /dev/null)" ]
11        then
12            cp -pr ${datadir}-template/* ${DATA_PATH}/${datadir#/*}/
13        fi
14    fi
15 done
```

Para ello recorreremos cada una de las instancias del volumen e iremos creando los directorios, solo si no existe el directorio. Seguido comprobamos que no exista un enlace simbólico, si de verdad no existe, copiaremos el contenido del directorio template dentro del directorio datadir.

En versiones antiguas de Naemon la contraseña del archivo httpasswd de thruk se guardaba en el archivo /etc/naemon, a través de nuestro script run comprobaremos que si es una versión antigua y se encuentra en dicha carpeta lo traslade al archivo /etc/thruk/httpasswd.

```
1 if [ -e /etc/naemon/httpasswd ]
2 then
3 echo "UPGRADE: Moving the httpasswd file to the new location..."
4 mv /etc/naemon/httpasswd /etc/thruk/httpasswd
5 # We assume the naemon password was set in an upgrade situation
6 touch /etc/thruk/._install_script_password_set
7 fi
```

Por defecto se asigna el usuario y contraseña **thrukadmin** dentro del fichero /etc/thruk/httpasswd.

Si no queremos que esa sea la contraseña asignada, podemos crear dentro del script una combinación para crear una contraseña aleatoria, para ello usaremos la variable `RANDOM_PASS` y `WEB_ADMIN_PASSWORD` donde se asignará dicha contraseña.

A través de la siguientes líneas realizaremos lo comentado:

```
1 RANDOM_PASS='date +%s | md5sum | base64 | head -c 8'
2 WEB_ADMIN_PASSWORD=${WEB_ADMIN_PASSWORD:-$RANDOM_PASS}
3 htpasswd -bc /etc/thruk/htpasswd thrukadmin ${WEB_ADMIN_PASSWORD}
4 echo "Set the thrukadmin password to: $WEB_ADMIN_PASSWORD"
5 touch /etc/thruk/._install_script_password_set
```

Ahora nos encargaremos de comprobar que si estamos actualizando desde un contenedor anterior, moveremos el archivo llamado **cgi.cfg** a una nueva localización. Es decir, moveremos desde la ruta `/etc/naemon/cgi.cfg` a `/etc/thruk/cgi.cfg`.

A continuación se creará una función para realizar la parada del servidor apache y matar el proceso Naemon.

Realizamos por tanto lo siguiente:

```
1 function salida_exitosa(){
2 /etc/init.d/apache2 stop
3 pkill naemon
4 exit $1
5 }
```

Hay que señalar que la imagen de phusion no funciona el comando **service naemon stop** por lo que habrá que realizar lo anterior.

Ahora nos aseguramos de dar los permisos de propiedad al volumen creado, es decir, a **data**, incluso si se cambian los UID o GID.

```
1 chown -R naemon:naemon /data/etc/naemon /data/var/log/naemon
2 chown -R www-data:www-data /data/var/log/thruk /data/etc/thruk
```

Asignamos el permiso 775 al directorio `/var/cache/naemon` ya que no es editable por otros usuarios que no sean naemon, lo que significa que la herramienta de configuración de thruk no puede escribir en el directorio para verificaciones de configuración.

```
1 chmod 775 /var/cache/naemon
```

Solo nos quedará iniciar los servicios de Naemon y Apache con:

```
1 service naemon start
2 service apache2 start
```

Anteriormente habíamos realizado una función para realizar una salida exitosa, ahora haremos uso de dicha función comprobando si naemon o apache está funcionando o no:

```
1 trap "salida_exitosa 0;" SIGINT SIGTERM
2
3 while true
4 do
5 service naemon status > /dev/null
6 if (( $? != 0 ))
7 then
8 echo "Naemon no longer running"
9 salida_exitosa 1
10 fi
11
12 /etc/init.d/apache2 status > /dev/null
13 if (( $? != 0 ))
14 then
15 echo "Apache no longer running"
16 salida_exitosa 2
17 fi
18 sleep 1
19 done
```

El comando **trap** se usa para especificar las acciones a realizar cuando se reciban las señales.

Con todo esto ya estaría nuestro script listo solo para realizar la construcción de nuestra imagen.

3.3.8.2. Creación de imagen a través de docker build

Una vez tengamos la forma de realizar la correspondiente ejecución pasaremos a ejecutar su comando correspondiente en **Docker** para crear la imagen con todo lo anterior explicado. Para ello a través de terminal estableceremos el siguiente comando:

```
1 $ docker build -t chui274/naemontfg .
```

A la hora de crear la imagen indicaremos que se introduzca todo el contenido que hay en la raíz a través del `.` y con la opción `-t` indicaremos que queremos etiquetar la imagen con el nombre `naemontfg` con el usuario `chui274` correspondiente, es decir, se creará un repositorio con el nombre `chui274/naemontfg`. Podemos asignarle el nombre que queramos al repositorio creado.

3.3.8.3. Ejecución de imagen

Para ejecutar la imagen creada solo tendremos que ejecutar el comando **docker run**. Para ello realizaremos la siguiente combinación de parámetros:

```
1 $ docker run --rm -it -p 80:80/tcp chui274/naemontfg
```

Donde le indicaremos con los parámetros asociados lo siguiente:

- **-rm** con esto hacemos que si existe el contenedor automáticamente lo borra.
- **-it** Permite el uso interactivo del contenedor.
- **-p** Aplica un puerto de escucha(en este caso se trata del puerto 80 a través de comunicación TCP)

Si accedemos a través del navegador a la dirección `http://localhost:80` accederemos a la interfaz de **Thruk** como sigue:

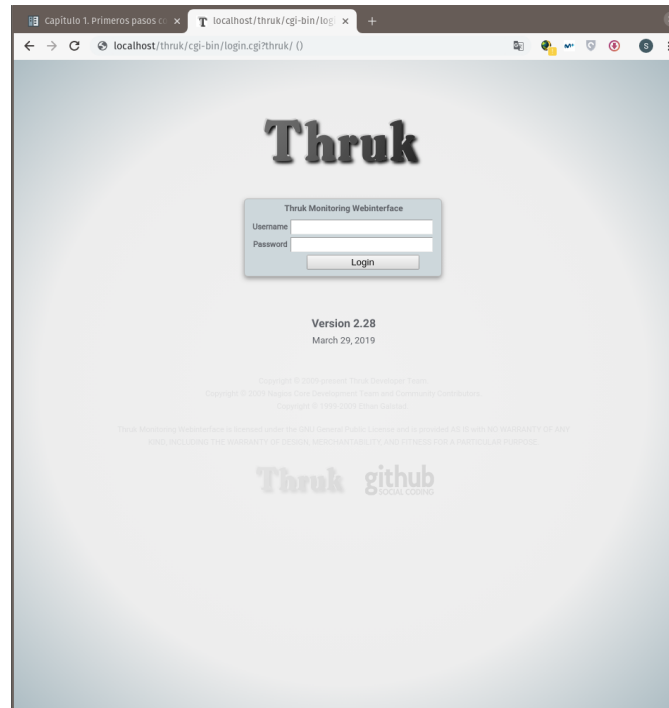


Figura 3.10: Ejecución de Docker

3.4. Orquestación de aplicaciones

Existen múltiples definiciones sobre el concepto de **orquestación de aplicaciones** pero de un modo simple, podemos definir la **orquestación de servicios o aplicaciones** como el uso de la automatización para la creación y composición de la arquitectura, herramientas y procesos utilizados por operadores humanos para entregar un servicio.

La **orquestación** aprovecha tareas automatizadas y procesos pre-definidos para permitir la creación de infraestructura complejas y para conseguir el aprovechamiento de los recursos de forma óptima y automatizada. Podemos considerar, a modo de analogía, el concepto de orquestación como un proceso y la automatización como una tarea.

De este modo, el objetivo principal de la orquestación consiste en la automatización de procesos orientados al despliegue y ciclo de vida de las aplicaciones o servicios. Y la automatización de procesos en los despliegues software se basan en el uso de algún tipo de software que facilite la instalación, configuración y mantenimiento del servicio o aplicación con la mínima intervención humana.

Existen dos tipos distintos de **orquestación** en base fundamentalmente a como se gestiona el escalado de recursos.

- **Orquestación estática:** El sistema requiere una configuración más manual de los recursos y no permite el escalado de forma muy eficiente.
- **Orquestación Dinámica:** Escala de forma sencilla y eficientes los recursos de las aplicaciones y servicios y el propio sistema toma ciertas decisiones de forma automática.

En este proyecto, se plantea una solución de **orquestación estática** a través de la herramienta que cuenta Docker, llamada **Docker Compose** que permite la orquestación estática orientada a un funcionamiento más centrado un solo servidor, esta nos permite a través de un **fichero YML**, la definición y ejecución de aplicaciones Docker en múltiples contenedores.

Antes de comenzar con la explicación de **Docker Compose** explicaremos que es el formato **YML** o más bien conocido como **YAML** (Yet Another Markup Language). [24]

YAML se trata de un formato de serialización de datos de forma que sea legible para el usuario. Se inspira en lenguajes como XML, C, Python, Perl. Normalmente lo encontraremos con la extensión **.yaml** o incluso **.yml**.



Figura 3.11: Logo de YAML

Existen unas **reglas** generales que deben cumplirse en un documento **YAML** las principales son las siguientes:

- Los datos de un documento YAML deben ser legibles, imprimibles y utilizando caracteres Unicode, UTF-8 ó UTF-16.
- Los comentarios se realizan utilizando el carácter `#` dentro de la línea que contiene el comentario.
- Los caracteres `,` y `;` deben ir seguidos de un espacio en blanco. De esta forma, se podrán representar valores que queramos que tengan esos caracteres.
- Los espacios en blanco están permitidos, pero no los tabuladores.
- Las listas comienzan por el caracter `“—”` con un valor por cada línea, aunque también se pueden utilizar corchetes `[]` poniendo los valores dentro de ellos separados por comas `“,”` junto con un espacio en blanco.
- Un vector estará formado por el par **clave/valor**, estando separados ambos por `“:”` poniendo uno por línea, aunque también podemos utilizar `“,”` poniendo cada uno de ellos dentro separados por comas `“,”` junto con un espacio en blanco.
- Se pueden utilizar caracteres de escape para representar caracteres especiales.

3.4.1. Uso de Docker Compose

Como se ha mencionado se utilizará dicha herramienta para la ejecución de varios contenedores de Naemon trabajando de forma distribuida, pudiendo crear e iniciar todos a la vez.

Para ello tenemos que definir un fichero `Dockerfile`, ya lo tenemos creado. Seguidamente crearemos un archivo llamado **`docker-compose.yml`** que se encargará de las ejecuciones del entorno.

Y para finalizar y reproducir la ejecución solo será necesario introducir a través de terminal desde la ruta en la que se encuentra el archivo de Docker Compose introducir:

```
1 $ docker-compose up
```

A continuación se comenta lo introducido en dicho archivo docker-compose.yml:

```
1 version: '3.7'
2 services:
3   naemon:
4     build: .
5
6   ports:
7     - "32768:80"
8   image: "chui274/naemontfg"
```

Con la etiqueta **version** especificaremos la versión de Compose e incluyendo la versión de Docker Engine, por lo que si asignamos **version: '3.7'** estamos indicando que queremos la versión 3.7 de Docker Compose y la versión 18.06.0+ de Docker Engine.

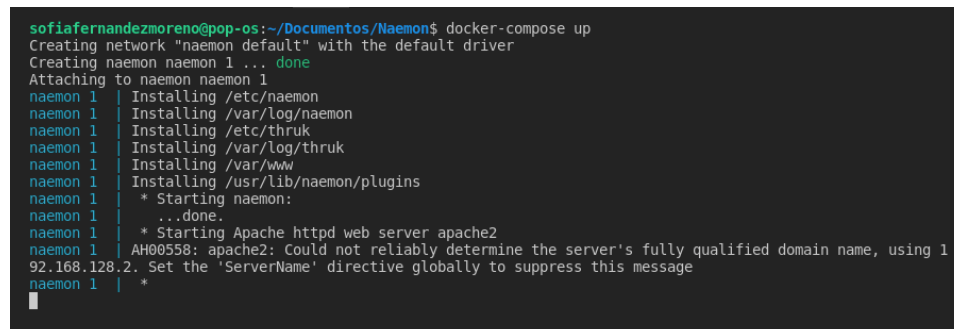
A través de la etiqueta **services** comenzaremos a definir los contenedores que queremos crear, comenzamos creando nuestro contenedor llamado **naemon**, para la construcción es necesario la etiqueta **build**.

La etiqueta **build** indicará la imagen de Dockerfile que vamos a usar, le indicamos **.** para que lea desde el archivo Dockerfile que se encontrará en el directorio principal.

Le indicaremos a través de la etiqueta **ports** que realiza la escucha de la interfaz de thruk a través del puerto 32768.

Ahora a través de la etiqueta **image** si hemos realizado la construcción bien con **build**, cogeremos la imagen desde **chui274/naemontfg**, que ha sido construida desde **.**

Realizando el comando anterior mencionado a través de terminal, tendremos lo siguiente:



```
sofiafernandezmoreno@pop-os:~/Documentos/Naemon$ docker-compose up
Creating network "naemon default" with the default driver
Creating naemon naemon 1 ... done
Attaching to naemon naemon 1
naemon 1 | Installing /etc/naemon
naemon 1 | Installing /var/log/naemon
naemon 1 | Installing /etc/thruk
naemon 1 | Installing /var/log/thruk
naemon 1 | Installing /var/www
naemon 1 | Installing /usr/lib/naemon/plugins
naemon 1 | * Starting naemon:
naemon 1 | ...done.
naemon 1 | * Starting Apache httpd web server apache2
naemon 1 | AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 1
naemon 1 | 92.168.128.2. Set the 'ServerName' directive globally to suppress this message
naemon 1 | *
```

Figura 3.12: Ejecución de Docker Compose

Ahora estamos a disposición de probar la dirección `http://localhost:32768/` para acceder a Naemon a través de Thruk.

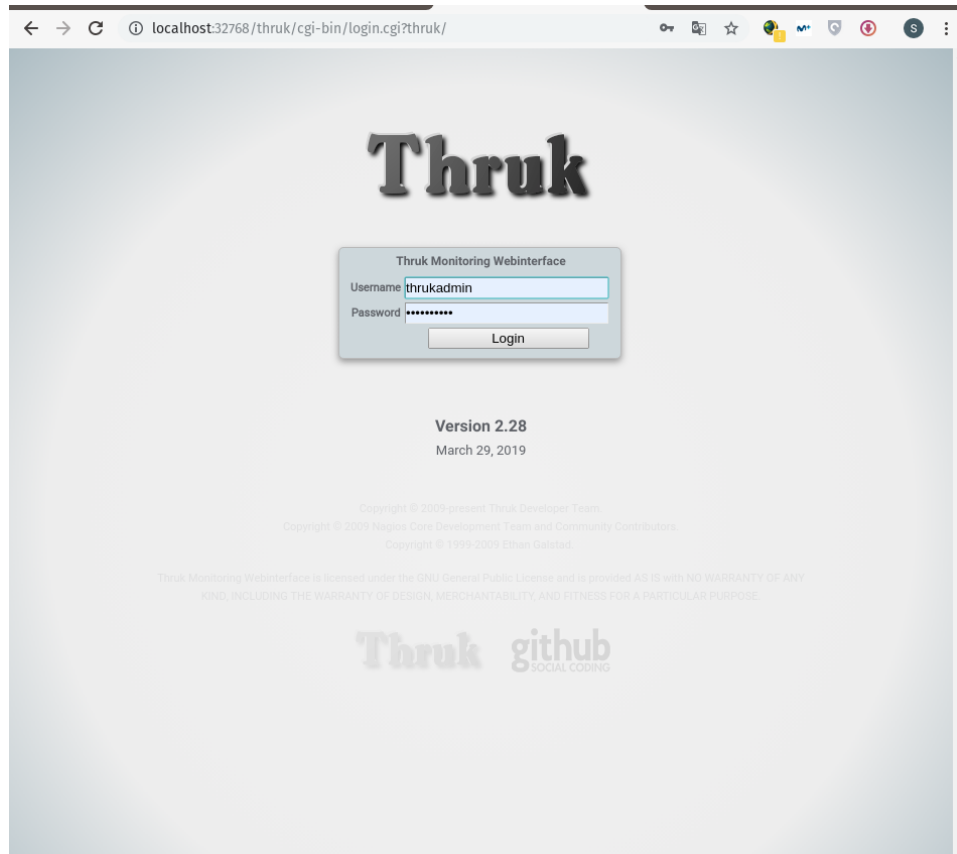


Figura 3.13: Ejecución de Thruk mediante Docker Compose

Capítulo 4

Pruebas de carga

4.1. Introducción

A la hora de trabajar con un sistema a un nivel de rendimiento estable y bueno es fundamental que se realicen pruebas al comienzo del inicio del desarrollo de su software. Al igual que en las pruebas funcionales, el coste de solucionar defectos se ve aumentado conforme más se tarde en detectarlos.[25]

Para que las pruebas sean lo más fiables posible, el entorno de prueba debe ser lo más parecido posible al de producción.

Las **pruebas de rendimiento** son un conjunto de pruebas que nos permiten medir la velocidad de ejecución de una serie de tareas en un sistema, bajo unas condiciones determinadas.

Las **pruebas de rendimiento** sirven, entre otras cosas, para:

- Demostrar que el sistema cumple los criterios de rendimiento.
- Validar y verificar atributos de la calidad del sistema: escalabilidad, fiabilidad, uso de los recursos.
- Comparar dos sistemas para saber cuál de ellos funciona mejor.
- Detectar cuellos de botella, es decir, medir qué partes del sistema o de carga de trabajo provocan que el conjunto rinda mal.

4.2. Tipos de pruebas de rendimiento

Podemos encontrarnos con las siguientes pruebas a la hora de probar el rendimiento de un sistema:

4.2.1. Prueba de carga

Se trata de una prueba que generalmente observa el comportamiento de una aplicación bajo una serie de peticiones. Por ejemplo puede ser el número esperado de usuarios concurrentes, utilizando la aplicación que realizan un número específico de transacciones, durante el tiempo que dura la carga.

Esta prueba puede mostrar los tiempos de respuesta de todas las transacciones importantes de la aplicación. Si también se monitorizan otros aspectos como la base de datos, el servidor de aplicaciones, etc., entonces esta prueba puede mostrar el cuello de botella en la aplicación.[26]

4.2.2. Prueba de estrés

Se utiliza como su propio nombre dice estresar la aplicación, es decir, que se encuentre en un estado forzado. Se va aumentando el número de usuarios que se agregan a la aplicación y se ejecuta una prueba de carga hasta que se rompe. Este tipo de prueba se realiza para determinar la solidez de la aplicación en los momentos de carga extrema. Esto ayuda a los administradores para determinar si la aplicación rendirá lo suficiente en caso de que la carga real supere a la carga esperada.[26]

4.2.3. Prueba de estabilidad (Soak Testing)

Se realiza para determinar si la aplicación es capaz de aguantar una carga concreta. El objetivo principal de este tipo de pruebas es verificar que no existen fugas de memoria o procesos que pierdan rendimiento transcurrido un cierto periodo de tiempo.[26]

4.2.4. Prueba de pico (Spike Testing)

En esta prueba, la aplicación se prueba con incrementos y decrementos extremos en la carga. Se realiza para estimar la debilidad de una aplicación.

Ayuda a evaluar el comportamiento del sistema de software en incrementos o disminuciones repentinos en la carga del usuario.[26]

4.3. Tipo de prueba seleccionada

Para este proyecto se ha elegido la opción de realizar pruebas de carga para poder simular el uso de concurrencia real en nuestro despliegue de Naemon.

Además usaremos este tipo de prueba puesto que queremos medir la *performance* de un sitio web, el cual mencionaremos más adelante y este tipo de prueba nos será de gran utilidad.

A continuación se introducirán una serie de herramientas para poder simular este tipo de prueba de carga, donde finalmente se escogerá entre todas una concreta.

4.3.0.1. Gatling

Se trata de una herramienta de código abierto y multiplataforma, es decir, podremos utilizarlo en cualquier dispositivo, por lo que permite gran libertad en cuanto a su uso. Además, ofrece una gestión óptima de los recursos del sistema frente a otras herramientas como JMeter[27]. Estas pruebas se pueden hacer sobre diversos protocolos como pueden ser HTTP (páginas web, servicios REST), FTP, sockets de web, etc.[28]



Figura 4.1: Gatling

Las pruebas de **Gatling** consistirán en simulaciones donde se realizarán flujos de uso de la web, y al final obtendremos un reporte detallado con los tiempos de respuesta, resultados de las peticiones, etc.

4.3.0.2. Locust

Locust [29] es una herramienta de prueba de carga de usuario distribuida y fácil de usar. Está diseñado para probar sitios web (u otros sistemas) y determinar cuántos usuarios concurrentes puede manejar un sistema.

La idea es que durante una prueba, un enjambre de langostas atacará el sitio web. El comportamiento de cada langosta (o usuario de prueba, si lo desea) se define por nuestra cuenta y el proceso de enjambre se supervisa desde una interfaz de usuario web en tiempo real. Esto ayudará a probar e identificar cuellos de botella en el código antes de permitir el ingreso de usuarios reales.



Figura 4.2: Locust

Locust está completamente basado en eventos y, por lo tanto, es posible admitir miles de usuarios concurrentes en una sola máquina. A diferencia de muchas otras aplicaciones basadas en eventos, no usa devoluciones de llamadas. En cambio, usa procesos ligeros, a través de *gevent*. Cada langosta pululando en su sitio se está ejecutando dentro de su propio proceso (o *greenlet*, para ser correcto). Esto permite escribir escenarios muy expresivos en Python sin complicar su código con devoluciones de llamadas.

4.3.0.3. Jmeter

Apache JMeter [30] es la única aplicación de escritorio en esta revisión. Tiene una interfaz gráfica de usuario fácil de usar, lo que hace que el desarrollo de pruebas y la depuración sean mucho más fáciles. Tiene una estructura modular, en la que el núcleo se extiende mediante complementos. Esto significa que todos los protocolos y características implementados son complementos que han sido desarrollados por Apache Software Foundation o colaboradores en línea.



Figura 4.3: JMeter

Tiene más funciones e integraciones, y también una base de usuarios más grande que cualquier otra herramienta de prueba de carga de código abierto.

El principal aspecto negativo de JMeter es que las configuraciones, incluida la lógica del escenario del usuario, están escritas en XML.

4.3.1. Alternativa elegida: Locust

Se ha elegido esta opción puesto que se trata de un programa multiplataforma por lo que podremos ejecutarlo en cualquier sistema operativo. Además de ofrecer una alta escalabilidad debido a la implementación que cuenta basada en eventos. Otra cosa a destacar, se trata de la escalabilidad en cuanto a la distribución de los agentes, permite incluir un número infinito de agentes.

Locust es altamente escalable debido a su implementación totalmente basada en eventos. Debido a estos hechos, Locust tiene una comunidad en rápido crecimiento, que prefieren este marco en lugar de JMeter (siendo ésta de las más populares).

Cuenta con una interfaz de usuario de comando y control basada en web, además de soporte para la generación de carga distribuida, lo que lo hace ser más actualizable y seguro para el futuro.

Permite además la descarga de los resultados en tipo CSV y JSON.

4.4. Instalación de Locust

Locust está disponible en **PyPI** y se puede instalar a través del comando `pip`.

```
1 $ apt-get install python-pip
2 $ pip install locustio
```

4.5. Funcionamiento de herramienta Locust

Para poder empezar a trabajar en Locust es necesario tener activado un servidor web, que en este caso será **Apache**. Una vez inicializado el servidor web de Apache, si no se encuentra activo podemos activarlo a través del comando:

```
1 $ service apache2 start
```

Activado el servidor web, debemos ejecutar **Locust** en el servidor Web Apache, para ello debemos configurar un archivo **locustfile.py** que contará con el código necesario para realizar las pruebas de carga necesarias, distribuyendo la prueba de rendimiento en diferentes máquinas para que se pueda crear más carga en la aplicación.[31]

4.6. Ejecución distribuida

La ejecución de una sola máquina no es suficiente para modelar o simular una carga, por lo que Locust admite la ejecución de pruebas de carga distribuidas en múltiples máquinas.

Para hacer esto, se debe iniciar una instancia de Locust en modo maestro usando el flag `-master`. Dicha instancia será la ejecutada por la interfaz web de Locust, donde se inicia la prueba y se muestran las estadísticas en vivo.

Para poder simular las cargas debemos de iniciar uno o varios nodos con el flag `-slave` para simular los esclavos, junto con el `-master-host` (especificando la IP o nombre del host del nodo maestro).

Una configuración común es ejecutar un solo maestro en una máquina, y luego ejecutar una instancia esclava por núcleo del procesador, en las máquinas esclavas.

Para poder establecer dicha configuración de forma automática a través del despliegue realizado en la tecnología **Docker**, debemos crear una nueva instancia dentro de nuestro archivo **docker-compose.yml**.

Para ello creamos las máquinas llamadas **locust-master** y **locust-worker** por las cuales queremos simular la carga maestro-esclavo a través de Docker.

En el archivo **docker-compose.yml** crearemos estas máquinas a través de la imagen **swernst/locusts**, la cual cuenta con la **última versión de Locust para Docker**, recogida del repositorio de **Docker Hub**. En esta imagen tendremos que la instancia del maestro se encarga de alojar una interfaz web, así como la coordinación con los esclavos. En este caso el esclavo se encargará de realizar los test y crear reportes de información al maestro de forma estadística.

Esta división permite escalar el número de esclavos para realizar pruebas de carga realmente grandes.

Ambas vinculan el directorio **scripts** en el contenedor a un subdirectorio en nuestra máquina host con el mismo nombre. Esto hará que nos permita eliminar los scripts que se quieran usar para dicha prueba y ponerlo a disposición de cualquier caso.

Tenemos que el servicio **locust-master** tiene un puerto definido, exactamente el puerto 8089, lo que hace que podamos tener acceso a la interfaz web de Locust desde ese servicio. Además el servicio **locust-worker** está tomando un parámetro de comando adicional que le dice que se conecta a master para la orquestación, esto es a través de la línea: **command: -master-host=locust-master**".

Esto redactado dentro de nuestro archivo **docker-compose.yml** quedará de la siguiente manera:

```
1
2 locust-master:
3 image: swernst/locusts
4 volumes:
5 - ./scripts:/scripts
6 ports:
7 - "8089:8089"
8 depends_on:
9 - wordpress
10 locust-worker:
11 image: swernst/locusts
12 command: "--master-host=locust-master"
13 volumes:
14 - ./scripts:/scripts
```

Donde la línea establecida en locust-master como:

```
1 depends_on:
2 - wordpress
```

Viene indicada para la siguiente explicación donde trabajaremos con el sistema final establecido, el cual será en **WordPress**, donde también será creada un servicio a través del archivo **docker-compose**.

A la hora de lanzar esta ejecución en el cual tengamos tres instancias trabajando, o lo que es lo mismo tres esclavos, debemos ejecutar la siguiente línea para lanzar el contenedor Docker:

```
1 $ docker-compose up --scale locust-worker=3
```

O simplemente si queremos un maestro y un esclavo haríamos el levantamiento del contenedor de la siguiente manera:

```
1 $ docker-compose up
```


4.7. Configuración Locustfile

Locust generará una hebra para cada usuario que está siendo simulado y cada usuario viene representado por una clase **locust**. Para empezar con la configuración de nuestro archivo **locustfile.py**, por el cual nos encargaremos de realizar las pruebas de carga correspondientes a nuestro sistema final, pero antes debemos explicar los atributos que vamos a utilizar para realizar estos test o pruebas:

- **TaskSet**: se trata de una colección de tareas, es decir, define el comportamiento del usuario. Para cada acción definida debe definirse la anotación **@task**.
- **HttpLocust**: utilizada para cargar la prueba de rendimiento de un sistema en el servidor. Representa un usuario el cual será atacado y que será probado con carga. Dicha clase crea un atributo cliente que se trata del cliente HTTP que se encarga de realizar el enlace entre la sesión y las peticiones.

Un ejemplo de un test simple sería el siguiente:

```
1 from locust import HttpLocust, TaskSet, task
2
3 class UserBehavior(TaskSet):
4     @task(2)
5     def root(self):
6         self.client.get('/')
7     @task(1)
8     def host(self):
9         self.client.get('/?p=1')
10 class WebsiteUser(HttpLocust):
11     task_set = UserBehavior
12     min_wait = 5000
13     max_wait = 9000
```

Donde se ha realizado la obtención de los datos de la página principal del sistema y también la obtención de los datos de un artículo concreto del sistema que en este caso ha sido WordPress.

Hay que especificar que en la clase Locust se permite especificar el tiempo de espera mínimo y máximo en milisegundos, por usuario simulador, entre la ejecución de las tareas, esto se realiza mediante las variables **min_wait** y **max_wait**.

Cabe mencionar que de forma predeterminada, el tiempo se elige aleatoriamente de forma uniforme entre dichas variables, pero se puede usar cualquier distribución de tiempo definida por el usuario configurando una variable **wait_function** en cualquier función arbitraria. Podemos realizar como ejemplo un tiempo de espera distribuido de forma exponencial con un promedio de 1 segundo, esto en código Python se establece de la siguiente forma:

```
1 class WebsiteUser(HttpLocust):
2     task_set = UserBehavior
3     wait_function = lambda self: random.expovariate(1)*1000
```

En el siguiente capítulo pondremos este archivo de configuración a prueba mediante el despliegue de un sistema WordPress, el cual monitorizar la generación de carga por parte de los clientes y de cómo sirve la carga el sistema, además de modelar la carga a partir de los datos recopilados durante la monitorización.

Capítulo 5

Pruebas de carga en sistema

En este capítulo profundizaremos sobre la monitorización de un sistema, en este caso supondrá un sistema en **WordPress**, este se trata de un sistema de gestión de contenidos.

A través de la herramienta **Naemon** y el framework de **Locust** queremos realizar el análisis de dicho sistema.

5.1. Despliegue de sistema a través del contenedor Docker

Como se realizó en capítulos anteriores para realizar el despliegue de Naemon y Locust en Docker, esta vez pasaremos a la realización del despliegue de WordPress. Para ello nos basaremos en dos instancias separadas: **una instancia para Wordpress y otra instancia separada para MySQL con un volumen montado para la persistencia de los datos de MySql.**

En él definimos dos **servicios**: **db** (correspondiente a MySQL) y **wordpress**. En el **primer servicio (db)** indicamos la imagen que queremos descargarnos (un MySQL versión 5.7) y algunos parámetros de configuración. Los más relevantes son la redirección de puertos (del 8081 del anfitrión al 3306 del huésped) y parámetros de la base de datos en sí (usuario, contraseña, etc).

El **segundo servicio es WordPress (wordpress)** y sigue un patrón parecido. Aquí, por ejemplo, indicamos que queremos usar el puerto 80 para acceder a nuestro WordPress. También especificamos que db es una dependencia de WordPress (este contenedor no puede funcionar sin el otro) esto lo haremos mediante la opción **depends on**. Y también indicamos parámetros adicionales como, por ejemplo, dónde está la base de datos y cómo acceder a ella. Todo esto vendrá definido en el archivo **docker-compose.yml** de la siguiente manera:

```
1 db:
2     image: mysql:5.7
3     volumes:
4     - db_data:/var/lib/mysql
5     restart: always
6     environment:
7         MYSQL_ROOT_PASSWORD: somewordpress
8         MYSQL_DATABASE: wordpress
9         MYSQL_USER: wordpress
10        MYSQL_PASSWORD: wordpress
11
12 wordpress:
13     depends_on:
14     - db
15     image: wordpress:latest
16     ports:
17     - "80:80"
18     restart: always
19     environment:
20         WORDPRESS_DB_HOST: db:3306
21         WORDPRESS_DB_USER: wordpress
22         WORDPRESS_DB_PASSWORD: wordpress
23         WORDPRESS_DB_NAME: wordpress
```

Si realizamos la puesta en marcha mediante el comando **docker-compose up** comenzará con la descarga de las imágenes de **WordPress y MySQL**, y de forma seguida pondrá en funcionamiento el sistema de **WordPress**.

Cuando termine la instalación del contenedor, accedemos al navegador e introducimos la siguiente dirección **http://localhost:80** o **http://localhost**, donde se lanzará la instalación correspondiente de **WordPress**.

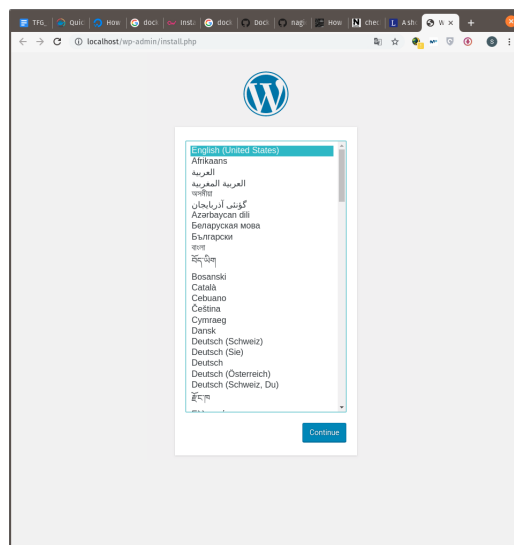


Figura 5.1: Instalación de WordPress

Realizado este paso solo quedará instalar **WordPress** con el blog correspondiente que se quiera ejecutar, en este caso se ha llamado al blog **Monitorización TFG**:

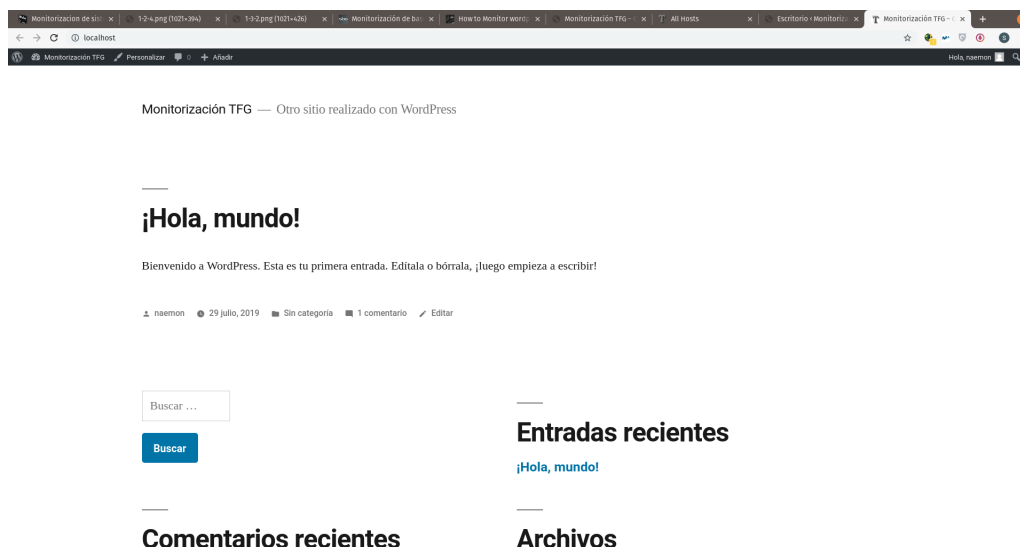


Figura 5.2: Pantalla principal Blog de WordPress

5.2. Enlazado con Locust

Como ya contamos con nuestro sistema montado sólo nos quedará realizar el enlazado con el **Framework de Locust**, para ello es necesario adaptar la configuración interna del archivo **locust.config.json**. Este archivo será añadido al fichero **docker-compose.yml** junto al **locustfile.py**.

Dicho archivo nos servirá para especificar la **URL raíz de la API** a la que se va a redirigir la **API de Locust**, junto con la lista de nombres de clase para las subclases de Locust que se usarán en la prueba.

El archivo **JSON** que tendremos que crear será algo como el siguiente:

```
1 {  
2   "target": "http://wordpress",  
3   "locusts": ["WebsiteUser"]  
4 }
```

Donde **target** corresponde a la dirección que vamos a capturar, especificamos **http://wordpress** ya que wordpress coincide con la etiqueta de creación de la imagen de WordPress, se podría poner **http://localhost** y funcionaría, pero esto garantiza que verdaderamente estamos estableciendo la relación con ese entorno, ya que a la hora de la realización se ha encontrado un problema con el paso de peticiones GET a través del servidor Apache y era necesario aplicar un cambio interno en la configuración de WordPress, estableciendo el puerto base como 80. Por lo que esta medida es garantizable y más rápida de realizar, ya que se está trabajando a través del despliegue de Docker.

La etiqueta **locusts** corresponde a la tarea o clase por la que vamos a ejecutar, en este caso es **WebsiteUser** que realiza toda la funcionalidad de nuestro archivo.

5.2.1. Creación de archivo Locustfile

Como se realizó en capítulos anteriores para poder empezar a trabajar con **Locust** es necesario la creación del archivo **locustfile.py** para poder realizar las distintas pruebas de carga. Para ello analizaremos la prueba de carga a la hora de acceder y salir del sistema WordPress, además realizaremos una prueba de carga para un artículo concreto, esto se realizará mediante el lanzamiento de la petición **GET** hacia la URL **/?p=1**.

En el caso del **login** y el **logout**, lanzaremos peticiones **POST**, ya que éste tipo de petición se utiliza para enviar una entidad a un recurso en específico, causando a menudo un cambio en el estado o efectos secundarios en el servidor.

El código del archivo **locustfile.py** quedará de la siguiente manera:

```
1 from locust import HttpLocust, TaskSet, task
2
3 def login(l):
4     l.client.post("/login", {"username":"naemon", "password":"naemon"})
5
6 def logout(l):
7     l.client.post("/logout", {"username":"naemon", "password":"naemon"})
8
9
10 class UserBehavior(TaskSet):
11     def on_start(self):
12         login(self)
13
14     def on_stop(self):
15         logout(self)
16     @task(2)
17     def root(self):
18         self.client.get('/')
19     @task(1)
20     def host(self):
21         self.client.get('/?p=1')
22 class WebsiteUser(HttpLocust):
23     task_set = UserBehavior
24     min_wait = 5000
25     max_wait = 9000
```

Si lanzamos en terminal el comando **docker-compose up** y lanzamos el enlace **http://localhost:8089/** ejecutaremos **Locust** a través del navegador.

También podemos especificar el número de instancias de esclavos a través del comando siguiente:

```
1
2 $ docker-compose up --scale locust-worker=3
```

5.3. Pruebas de carga con sistema WordPress

Una vez contamos con los dos archivos anteriormente mencionados, pasamos a la ejecución de la prueba de carga, para ello accedemos al enlace `http://localhost:8089` donde nos aparecerá la pantalla principal de **Locust** donde debemos especificar el **número de usuarios a simular (Number of users simulate)** y el **Hatch rate**. El valor de **Hatch rate** representa por cada segundo, cuántos usuarios se agregarán a los usuarios actuales hasta la cantidad total de usuarios. Cada hatch realizado Locust llama a la función `on_start` si existe.

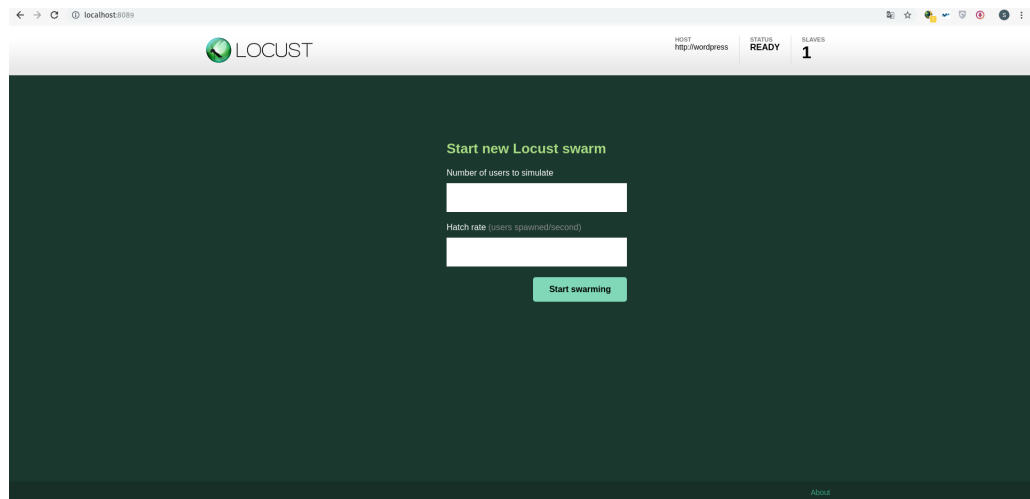


Figura 5.3: Interfaz Locust

Si ejecutamos como Number of users con valor 10 y el Hatch rate con valor 1 tenemos que cuando se inicie la prueba de carga con esta configuración, Locust generará 1 nuevo usuario por cada segundo hasta que alcance el número total de usuarios a simular (que en este caso es 10). Cuando alcanza el número de usuarios, la estadística se restablecerá.

Esta explicación viene representada de forma gráfica de la siguiente manera:

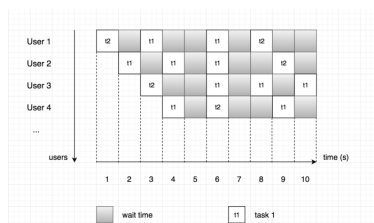


Figura 5.4: Realización de tareas concurrentes

Después de establecer estos valores, se podrá ver el resultado de la prueba en tiempo real:

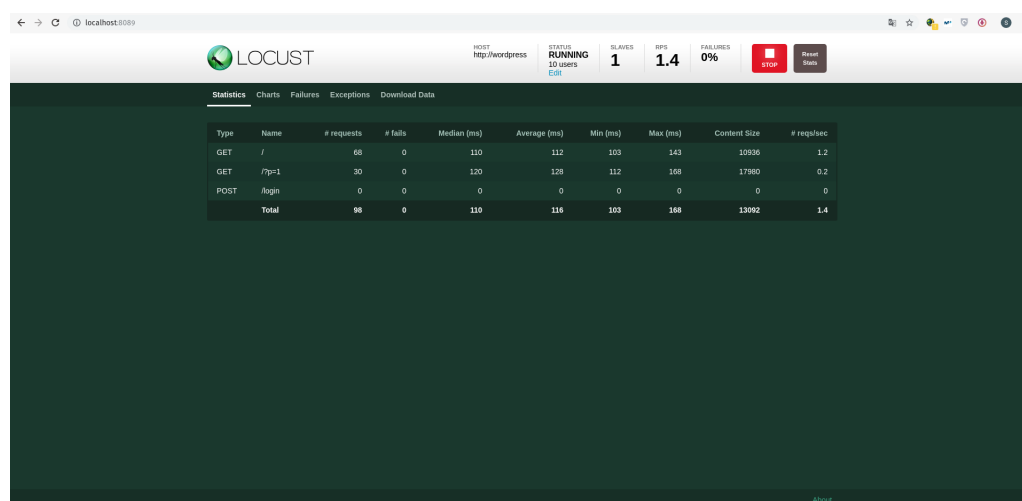


Figura 5.5: Pruebas de tareas concurrentes

Una vez que se ha alcanzado el número deseado de solicitudes, se puede detener la herramienta **Locust** desde la interfaz web. También permite restablecer las estadísticas o ejecutar una nueva prueba o test. Además de las estadísticas, puede mirar los **gráficos** (segunda pestaña) o cualquier **fallo o excepción que se haya recibido** (tercera, cuarta pestaña), así como **descargar los resultados de las pruebas como archivos CSV** (la última pestaña).

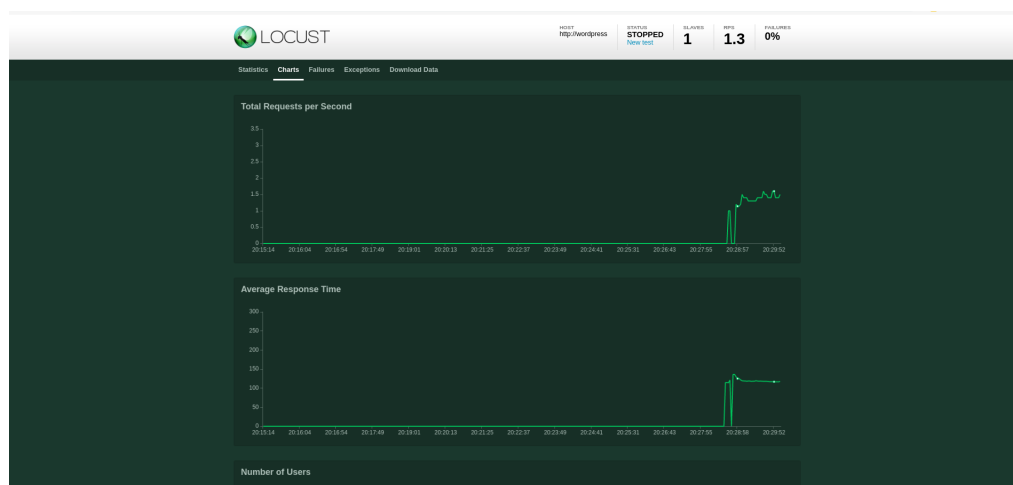


Figura 5.6: Gráfico generado en las ejecuciones

Básicamente, las **solicitudes por segundo**, es decir, el **rendimiento** indican la cantidad de transacciones por segundo que la aplicación puede realizar.

El **tiempo de respuesta** es la cantidad de tiempo desde el momento en que un usuario envía una solicitud hasta el momento en que su aplicación indica que la solicitud se ha completado.

En el gráfico anterior se puede ver la correlación entre los **tiempos de respuesta y el rendimiento**. El **rendimiento** general tiende a disminuir a medida que aumenta el tiempo de respuesta para una transacción promedio. El **motivo** es que después de enviar la primera solicitud, **Locust** debe esperar hasta que la solicitud se complete o se procese para enviar la segunda solicitud.

5.4. Pruebas de monitorización en Naemon

En esta sección vamos a empezar con la **puesta en marcha de la herramienta Naemon a través del análisis del sistema realizado**, es decir, el sistema WordPress. Como se señaló en el capítulo 2, donde especificábamos la carpeta de configuración de los distintos equipos y servicios a monitorizar, esta es la carpeta **/etc/naemon/conf.d**, en esta carpeta aplicaremos la configuración correspondiente para analizar el sistema WordPress.

Por defecto, Naemon configura la siguiente estructura de archivos y carpetas dentro de la carpeta anterior:

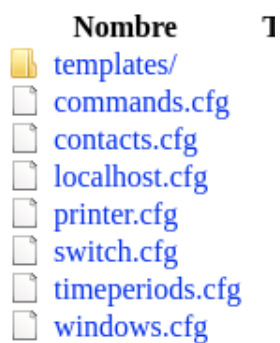


Figura 5.7: Contenido de /etc/naemon/conf.d

Dentro de la carpeta **templates** encontramos el siguiente contenido:

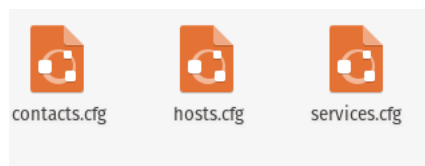


Figura 5.8: Contenido de /etc/naemon/conf.d/templates

Dentro de la carpeta **templates** podemos encontrar plantillas de ejemplos de hosts y services, donde el contenido de estos lo vamos a aplicar en la creación de nuestros archivos de configuración.

El archivo **localhost.cfg** es una plantilla de ejemplo del análisis de un servidor linux, tomaremos de ejemplo este para realizar las diferentes configuraciones, para poder introducir nuestros nuevos archivos de configuración, debemos modificar la imagen que creamos anteriormente para desplegar Naemon, donde incluiremos las siguientes líneas, donde añadiremos un archivo de incorporación de hosts y de servicios, además de eliminar dicho archivo **localhost.cfg**:

```
1 RUN rm -rf /etc/naemon/conf.d/localhost.cfg
2 ADD definition/wordpresshosts.cfg /etc/naemon/conf.d/wordpresshosts.
  cfg
3 ADD definition/wordpressservices.cfg /etc/naemon/conf.d/
  wordpressservices.cfg
```

A continuación se muestra el **contenido** de los dos nuevos archivos:

```
define host {
    host_name      wordpress
    alias          wordpress
    address        127.0.0.1
    use            linux-server
}
```

Figura 5.9: Contenido del archivo de configuración del host

```
# Define a service to "ping" the local machine
define service {
    service_description    PING
    host_name              wordpress
    use                    local-service          ; Name of service template to use
    check_command          check_ping!100.0,20%!500.0,60%
}

# Define a service to check the number of currently logged in
# users on the local machine. Warning if > 20 users, critical
# if > 50 users.
define service {
    service_description    Current Users
    host_name              wordpress
    use                    local-service          ; Name of service template to use
    check_command          check_local_users!20!50
}

# Define a service to check the number of currently running procs
# on the local machine. Warning if > 250 processes, critical if
# > 400 users.
define service {
    service_description    Total Processes
    host_name              wordpress
    use                    local-service          ; Name of service template to use
    check_command          check_local_procs!250!400!RSZDT
}

# Define a service to check the load on the local machine.
define service {
    service_description    Current Load
    host_name              wordpress
    use                    local-service          ; Name of service template to use
    check_command          check_local_load!5.0,4.0,3.0!10.0,6.0,4.0
}
```

Figura 5.10: Contenido del archivo de configuración del services

```
# Define a service to check the swap usage the local machine.
# Critical if less than 10% of swap is free, warning if less than 20% is free
define service {
    service_description    Swap Usage
    host_name              wordpress
    use                    local-service          ; Name of service template to use
    check_command          check_local_swap!20!10
}

# Define a service to check SSH on the local machine.
# Disable notifications for this service by default, as not all users may have SSH enabled.
define service {
    service_description    SSH
    host_name              wordpress
    use                    local-service          ; Name of service template to use
    check_command          check_ssh
    notifications_enabled  0
}

# Define a service to check HTTP on the local machine.
# Disable notifications for this service by default, as not all users may have HTTP enabled.
define service {
    service_description    HTTP
    host_name              wordpress
    use                    local-service          ; Name of service template to use
    check_command          check_http!-u /naemon/
    notifications_enabled  0
}
```

Figura 5.11: Contenido del archivo de configuración del services

En los siguientes apartados, se explicará las definiciones que se han realizado.

5.4.1. Definición del host

A través del archivo llamado con el nombre **wordpresshosts.cfg** se ha realizado la definición del host correspondiente al sistema wordpress, donde a través de:

- La etiqueta **host_name**, asigna el nombre del host, el cual se ha designado como **wordpress**.
- La etiqueta **alias**, asigna un alias al host, el cual se ha designado como **wordpress**.
- La etiqueta **address**, se encarga de establecer la dirección IP donde se encontrará el host ubicado. Estableceremos **127.0.0.1** que será nuestro **localhost**
- La etiqueta **use**, esta etiqueta se utiliza para usar una plantilla concreta, en este caso usaremos la creada por Naemon, la llamada **linux-server**, que hablaremos de ella a continuación.

El host utilizado **linux-server** se trata de un host creado por Naemon, que no contiene funcionalidad, pero utilizaremos como plantilla para poder trabajar, ya que este se encarga de definir el tiempo por el cual va a estar trabajando el host, además de comprobar en todo momento si está en activo, mediante el plugin **check-host-alive**. Este host se basa en **generic-host**, ésta se trata de otra plantilla, a continuación mostramos el contenido de cada plantilla de los host:

```

1
2 define host {
3     name                                generic-host
4     event_handler_enabled                1
5     flap_detection_enabled                1
6     notification_period                   24x7
7     notifications_enabled                 1
8     process_perf_data                     1
9     register                             1
10    retain_nonstatus_information           1
11    retain_status_information              1
12 }
13
14
15 define host {
16     name                                linux-server
17     use                                  generic-host
18     check_command                         check-host-alive
19     check_interval                         5
20     check_period                           24x7
21     contact_groups                         admins
22     max_check_attempts                     10
23     notification_interval                  120
24     notification_options                   d,u,r
25     notification_period                    workhours
26     register                             1
27     retry_interval                         1
28 }
```

Las directivas de la definición de los host se explicaron en el capítulo de Estado de Arte 2, en este caso el funcionamiento concreto que se ha establecido para el host **generic-host** es el siguiente:

- A través de la directiva **event_handler_enabled** a 1 hemos habilitado el controlador de eventos del host.
- A través de la directiva **flap_detection_enabled** a 1 hemos habilitado la detección de flaps del host.
- A través de la directiva **notification_period** se ha indicado que esté dispuesto 24h a la semana.
- A través de la directiva **notifications_enabled** a 1 hemos habilitado las notificaciones del host.
- A través de la directiva **process_perf_data** a 1 habilita el procesamiento de datos de rendimiento.
- A través de la directiva **register** a 1 hace que se registre el host.
- A través de la directiva **retain_nonstatus_information** a 1 habilita la retención de información sin estado.
- A través de la directiva **retain_status_information** a 1 habilita la retención de información de estado.

El funcionamiento concreto que se ha establecido para el host **linux-server** es el siguiente:

- A través de la directiva **use** hace que herede la información de generic-host.
- A través de la directiva **check_command** se intentará hacer un ping al host para ver si está vivo o activado.
- A través de la directiva **check_interval** se establecerá el valor 5 como unidad de tiempo entre las distintas comprobaciones realizadas.
- A través de la directiva **check_period** se activará una comprobación de 24h a la semana.
- A través de la directiva **contact_groups** se especificará que el grupo al que se notificará es al creado llamado admins.
- A través de la directiva **max_check_attempts** con valor 10, indicará que realizará diez veces el intento del comando de verificación del host.

- A través de la directiva **notifications_enabled** a 1 hemos habilitado las notificaciones del host.
- A través de la directiva **notification_period** se ha indicado que esté dispuesto en horas laborables.
- A través de la directiva **register** a 1 hace que se registre el host.

5.4.1.1. Host utilizado en la prueba de monitorización del sistema

En la figura 5.9 se muestra la configuración del host que se va a utilizar en nuestra prueba, para ello aplicaremos el uso de la plantilla ya creada y explicada **linux-server**, lo único que tendremos que especificar para la creación de este nuevo host es la **dirección IP** desde donde se va encontrar el host creado y el nombre del nombre, el cual llamaremos como el sistema utilizado, es decir, **wordpress**.

Si accedemos a Naemon a través de la GUI de Thruk, podremos ver reflejado la creación de este nuevo host. La información establecida del host vendrá reflejado en la figura 5.12.

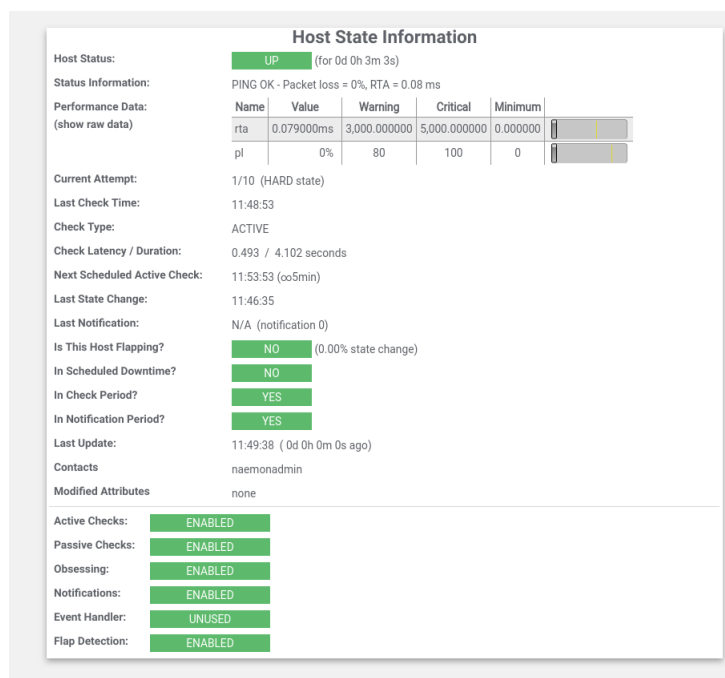


Figura 5.12: Información del host en Naemon

Como podemos observar el host se encuentra activo, ya que su estado es **UP**, además al indicarle que heredase las directivas de linux-server aparecerán también reflejadas en la información anteriormente mostrada.

En este caso a la hora de ejecutar el **PING**, servicio que explicaremos en el apartado siguiente, hacia este host, podemos ver que lo realiza de forma correcta, por lo que no se genera pérdida de datos ni paquetes, ya que además en la información mostrada se puede apreciar ese dato. Además mostrará el valor del **RTA** o lo que es lo mismo “*round trip average*”, que medirá la velocidad de transferencia de los datos, siendo este un valor bastante reducido como se muestra en la figura 5.12.

5.4.2. Definición de los servicios

A través de este archivo, definiremos los **servicios** que usaremos en los **chequeos** o comprobaciones. Se define la métrica o el servicio a monitorizar sobre el que se ejecuta. Como en el caso del host se ha establecido el uso de la plantilla aportada por Naemon, es decir, se ha utilizado la plantilla **local-service**.

```

1
2 define service {
3     name                                generic-service
4     active_checks_enabled              1
5     check_freshness                    0
6     check_interval                      10
7     check_period                       24x7
8     contact_groups                     admins
9     event_handler_enabled              1
10    flap_detection_enabled              1
11    is_volatile                         0
12    max_check_attempts                  3
13    notification_interval                60
14    notification_options                 w,u,c,r
15    notification_period                  24x7
16    notifications_enabled                1
17    obsess_over_service                  1

```

```

18     passive_checks_enabled      1
19                                ;
19     process_perf_data          1
20                                ;
20     register                   1
21                                ;
21     retain_nonstatus_information 1
22                                ;
22     retain_status_information   1
23                                ;
23     retry_interval             2
24                                ;
24 }
25
26
27 define service {
28     name                        local-service
29                                ;
29     use                        generic-service
30                                ;
30     check_interval             5
31                                ;
31     max_check_attempts         4
32                                ;
32     register                   1
33                                ;
33     retry_interval             1
34                                ;
34 }

```

Las directivas de la definición de los servicios se explicaron en el capítulo de Estado de Arte 2, como se ha realizado para la definición del host, a continuación se explicará el funcionamiento a seguir del servicio **generic-service** y **local-service**.

Para **generic-service** las directivas de funcionamiento establecidas son las siguientes:

- **is_volatile**: lo establecemos a 0 para indicar que no será un servicio volátil, es decir, no queremos que se pierda o se elimine la información recogida por el servicio.
- **max_check_attempts**: se utiliza para definir la cantidad de veces que Naemon volverá a intentar el comando de verificación de servicio si devuelve cualquier estado que no sea OK. Por lo que esto lo realizará una cantidad de tres veces.
- **check_interval**: se establecerá el valor 10 como unidad de tiempo entre las distintas comprobaciones realizadas.
- **retry_interval**: se establecerá el valor 2 como unidad de tiempo a esperar antes de programar una nueva verificación del servicio.

- **active_checks_enabled**: habilita las comprobaciones activas del servicio.
- **passive_checks_enabled**: habilita las comprobaciones pasivas del servicio.
- **check_period**: se activará una comprobación de 24h a la semana.
- **obsess_over_service—obsess** : habilita las comprobaciones para el servicio estarán sobrecargadas con el uso del comando `ocsp_command`.
- **check_freshness**: deshabilita las comprobaciones de actualización ya que esto puede suponer una carga sobre el servicio.
- **event_handler_enabled**: habilita el controlador de eventos de servicio.
- **flap_detection_enabled**: habilita la detección de flaps de servicio.
- **process_perf_data**: habilita el procesamiento de datos de rendimiento.
- **retain_status_information**: habilitar la retención de información de estado.
- **retain_nonstatus_information**: habilitar la retención de información sin estado.
- **notification_interval**: esperará 60 minutos para volver a notificar a un contacto que este servicio todavía está en un estado no correcto.
- **notification_period**: se ha indicado que esté dispuesto 24h a la semana.
- **notification_options**: se enviarán notificaciones en caso de estado de ADVERTENCIA, DESCONOCIDO, CRÍTICO Y RECUPERACIÓN.
- **notifications_enabled**: habilitar notificaciones de servicio.
- **contact_groups**: notificará a los grupos administrados, identificados como **admins**.

Para **local-service** las directivas de funcionamiento establecidas son las siguientes:

- **use:** indicará que heredará las directivas de generic-service.
- **max_check_attempts:** se utiliza para definir la cantidad de veces que Naemon volverá a intentar el comando de verificación de servicio si devuelve cualquier estado que no sea OK. Por lo que esto lo realizará una cantidad de cinco veces.
- **check_interval:** se establecerá el valor 4 como unidad de tiempo entre las distintas comprobaciones realizadas.
- **retry_interval:** se establecerá el valor 1 como unidad de tiempo a esperar antes de programar una nueva verificación del servicio.
- A través de la directiva **register** a 1 hace que se registre el servicio.

5.4.2.1. Servicios utilizados en la prueba de monitorización del sistema

En las figuras 5.10 y 5.11 se muestran la configuración de los servicios que se van a utilizar en nuestra prueba, para ello aplicaremos el uso de la plantilla ya creada y explicada **local-service** y **generic-service**.

Si accedemos a **Naemon** a través de la **GUI de Thruk**, podremos ver reflejada la creación de estos servicios. La información establecida del host con cada servicio vinculado vendrá reflejado en la figura 5.13.

Service	Status	Time	Details
Current Load	OK	12:05:47	OK - load average: 0.97, 1.05, 1.27
Current Users	OK	12:05:29	USERS OK - 0 users currently logged in
HTTP	OK	12:03:09	HTTP OK: HTTP/1.1 302 Found - 496 bytes in 0.001 second response time
PING	OK	12:03:03	PING OK - Packet loss = 0%, RTT = 0.08 ms
SSH	CRITICAL	12:04:22	connect to address 127.0.0.1 and port 22: Connection refused
Swap Usage	OK	12:05:09	SWAP OK - 100% free (3999 MB out of 3999 MB)
Total Processes	OK	12:06:35	PROCS OK: 26 processes with STATE = RSDT

Figura 5.13: Información del host con cada servicio vinculado en Naemon

A continuación vamos a hablar en detalle de cada servicio definido en el archivo **wordpressservices.cfg**.

5.4.2.1.1. Servicio PING A través de la directiva `check_command` haremos uso del plugin **check_ping**, por el cual hacemos uso de la llamada PING para comprobar las estadísticas de conexión del host.

Para este plugin se establecerá el **umbral** `<rta>,<pl>` donde `<rta>` será el tiempo o la velocidad de transferencia media en "ms" que activará un estado de **WARNING(advertencia)** o **CRITICAL(crítico)**, además `<pl>` es el porcentaje de pérdida de paquetes para activar un estado de alarma.

El uso de este comando suele ser de la siguiente forma:

```
1 check_ping -H <host_address> -w <wrta>,<wpl>% -c <crta>,<cpl>%  
2 [-p packets] [-t timeout] [-4|-6]
```

La función de `check_command` aplicará dicho comando plugin con la siguiente forma de definición:

```
1 check_ping!100.0,20%!500.0,60%
```

Donde se asume que:

- Para el estado **WARNING**: `<wrta>` es 100.0 y `<wpl>` es 0,20.
- Para el estado **CRITICAL**: `<crta>` es 500.0 y `<cpl>` es 0,60

Si este comando se accediera a través de una terminal o línea de comandos de forma expandida, se escribiría de la siguiente forma:

```
1 /usr/lib/naemon/plugins/check_ping -H 127.0.0.1 -w 100.0,20% -c  
500.0,60% -p 5
```

Donde se establece el host y el número de paquetes a enviar que si no se asume nada se toma por defecto el valor 5.

Este plugin utiliza el comando **ping** para que el host especificado busque la pérdida de paquetes según un porcentaje y el promedio de ida y vuelta en milisegundos.

Si accedemos a la **GUI de Thruk** podemos observar el registro de información del rendimiento, obteniéndose los siguientes valores:

```
1 rta=0.087000ms;100.000000;500.000000;0.000000 p1=0%;20;60;0
```

Por lo que el estado que obtendremos será **OK**, ya que no se ha perdido ningún paquete y los valores del **RTA** son tan pequeños que no se encuentran en el estado de **WARNING** ni **CRITICAL**.

5.4.2.1.2. Servicio para comprobar usuarios actuales registrados

A través de la directiva **check_command** haremos uso del siguiente plugin **check_local_users**, con este plugin comprobaremos el número de usuarios que se encuentran registrados en el servidor o host y genera un error si el número excede los umbrales especificados.

Para este plugin se establecerá el **umbral para warning y critical**.

El uso de este comando suele ser de la siguiente forma:

```
1 check_local_users -w <users> -c <users>
```

La función de **check_command** aplicará dicho comando plugin con la siguiente forma de definición:

```
1 check_local_users!20!50
```

Donde se asume que:

- Para el estado **WARNING**: se ha establecido un límite de 20 usuarios.
- Para el estado **CRITICAL**: se ha establecido un límite de 50 usuarios.

Si este comando se accediera a través de una terminal o línea de comandos de forma expandida, se escribiría de la siguiente forma:

```
1 /usr/lib/naemon/plugins/check_users -w 20 -c 50
```

Si accedemos a la **GUI de Thruk** podemos observar el registro de información del rendimiento, obteniéndose los siguientes valores:

```
1 USERS OK - 0 users currently logged in
2
3 users=0;20;50;0
```

Por lo que el estado que obtendremos será **OK**, ya que no se ha excedido los valores asignados en cuanto a usuarios conectados.

5.4.2.1.3. Servicio para comprobar total de procesos ejecutados

A través de la directiva **check_command** haremos uso del siguiente plugin **check_local_procs**, con este plugin comprobaremos todos los procesos y generaremos estados de ADVERTENCIA o CRÍTICOS si la métrica especificada está fuera de los rangos de umbral requeridos. El valor predeterminado de la métrica es el número de procesos. Los filtros de búsqueda se pueden aplicar para limitar los procesos a verificar.

El uso de este comando suele ser de la siguiente forma:

```
1 check_local_procs -w <range> -c <range> [-m metric] [-s state] [-p
   ppid]
2 [-u user] [-r rss] [-z vsz] [-P %cpu] [-a argument-array]
3 [-C command] [-k] [-t timeout] [-v]
```

La función de **check_command** aplicará dicho comando plugin con la siguiente forma de definición:

```
1 check_local_procs!250!400!RSZDT
```

Donde se asume que:

- Para el estado **WARNING**: se ha establecido un límite de 250 procesos.
- Para el estado **CRITICAL**: se ha establecido un límite de 400 procesos.
- Se ha establecido el estado **RSZDT**, el cual significa lo siguiente: **R**:ejecutable, **S**:esperando de forma dormida, **Z**:proceso zombie que termina pero no se repite por su padre, **D**: ininterrumpible, **T**:parado por una señal de control.

Si este comando se accediera a través de una terminal o línea de comandos de forma expandida, se escribiría de la siguiente forma:

```
1 /usr/lib/naemon/plugins/check_procs -w 250 -c 400 -s RSZDT
```

Si accedemos a la **GUI de Thruk** podemos observar el registro de información del rendimiento, obteniéndose los siguientes valores:

```
1 PROCS OK: 26 processes with STATE = RSZDT
2
3 procs=26;250;400;0;
```

Por lo que el estado que obtendremos será **OK**, ya que no se ha excedido los valores asignados en cuanto a procesos creados.

5.4.2.1.4. Servicio para comprobar la carga actual A través de la directiva `check_command` haremos uso del plugin `check_local_load`, con este plugin comprobaremos el promedio de carga actual del sistema.

El uso de este comando suele ser de la siguiente forma:

```
1 check_local_load [-r] -w WLOAD1,WLOAD5,WLOAD15 -c CLOAD1,CLOAD5,
   CLOAD15 [-n NUMBER_OF_PROCS]
```

- Donde **-r** se encarga de dividir el porcentaje de carga por el número de CPUs, pero esto solo cuando sea posible.
- Notificará con el estado de **ADVERTENCIA** si el promedio de carga excede el valor asignado en `WLOADn`
- Notificará con el estado de **CRÍTICO** si el promedio de carga excede el valor asignado en `CLOADn`

La función de `check_command` aplicará dicho comando plugin con la siguiente forma de definición:

```
1 check_local_load!5.0,4.0,3.0!10.0,6.0,4.0
```


Donde se asume que:

- Para el estado **WARNING**: se ha establecido las cargas de 5.0, 4.0 y 3.0
- Para el estado **CRITICAL**: se ha establecido las cargas de 10.0, 6.0 y 4.0

Si este comando se accediera a través de una terminal o línea de comandos de forma expandida, se escribiría de la siguiente forma:

```
1 /usr/lib/naemon/plugins/check_load -w 5.0,4.0,3.0 -c 10.0,6.0,4.0
```

Si accedemos a la **GUI de Thruk** podemos observar el registro de información del rendimiento, obteniéndose los siguientes valores:

```
1 OK - load average: 0.35, 0.64, 0.68
2
3 load1=0.350;5.000;10.000;0; load5=0.640;4.000;6.000;0; load15
  =0.680;3.000;4.000;0;
```

Por lo que el estado que obtendremos será **OK**, ya que no se ha excedido los valores asignados en cuanto a carga ejecutada.

5.4.2.1.5. Servicio para comprobar el uso de intercambio A través de la directiva `check_command` haremos uso del plugin `check_local_swap`, con este plugin comprobaremos el espacio de intercambio en la máquina local.

El uso de este comando suele ser de la siguiente forma:

```
1 check_swap [-av] -w <percent_free>% -c <percent_free>%
2 -w <bytes_free> -c <bytes_free> [-n <state>]
```

- Notificará con el estado de **ADVERTENCIA** si hay menos del porcentaje de espacio de intercambio libre o si hay menos bytes de espacio de intercambio libres.
- Notificará con el estado de **CRÍTICO** si hay menos del porcentaje de espacio de intercambio libre o si hay menos bytes de espacio de intercambio libres.

La función de **check_command** aplicará dicho comando plugin con la siguiente forma de definición:

```
1 check_local_swap!20!10
```

Donde se asume que:

- Para el estado **WARNING**: se ha establecido a 20 el límite
- Para el estado **CRITICAL**: se ha establecido a 10 el límite

Si este comando se accediera a través de una terminal o línea de comandos de forma expandida, se escribiría de la siguiente forma:

```
1 /usr/lib/naemon/plugins/check_swap -w 20 -c 10
```

Si accedemos a la **GUI de Thruk** podemos observar el registro de información del rendimiento, obteniéndose los siguientes valores:

```
1 SWAP OK - 100% free (3999 MB out of 3999 MB)
2
3 swap=3999MB;0;0;0;3999
```

Por lo que el estado que obtendremos será **OK**, ya que no se ha excedido los valores asignados, esto es debido a que nos encontramos en una máquina local sin tener a cargo una máquina Windows o Linux activada en el host.

5.4.2.1.6. Servicio para comprobar el protocolo SSH A través de la directiva **check_command** haremos uso del plugin **check_SSH**, con este plugin comprobaremos **el protocolo SSH**. Intentará conectarse a un servidor SSH en el servidor y puerto especificados

El uso de este comando suele ser de la siguiente forma:

```
1 check_ssh [-4|-6] [-t <timeout>] [-r <remote version>] [-p <port>] <host>
```

- Donde **-t** define los segundos antes del tiempo de espera de la conexión por defecto es 10.
- Donde **-r** alerta si la cadena no coincide con la versión esperada del servidor.
- Donde **-p** define el puerto a especificar.
- Se debe establecer el host por el cual se comprobará la conexión SSH.

La función de **check_command** aplicará dicho comando plugin con la siguiente forma de definición:

```
1 check_ssh
```

Si este comando se accediera a través de una terminal o línea de comandos de forma expandida, se escribiría de la siguiente forma:

```
1 /usr/lib/naemon/plugins/check_ssh 127.0.0.1
```

Si accedemos a la **GUI de Thruk** podemos observar el registro de información del rendimiento, obteniéndose los siguientes valores:

```
1 connect to address 127.0.0.1 and port 22: Connection refused
```

Esto es debido a que no se dió acceso al puerto 22 a la hora de realizar la creación del **Dockerfile**. Esto queda arreglado haciendo una llamada a **EXPOSE 22** en el archivo **Dockerfile**.

Y además añadiendo un nuevo servicio en **docker-compose.yml** llamado como el propio protocolo **ssh**. Esto quedará reflejado de la siguiente forma en dicho archivo:

```

1      ssh:
2      image: "jdeathe/centos-ssh:2.2.1"
3      volumes:
4      - "wp_html:/var/www/html"
5      - "wp_ssh_keys:/etc/ssh"
6      ports:
7      - "80:22"
8      environment:
9      SSH_USER: "wordpress"
10     SSH_USER_ID: "65534:65534"
11     SSH_USER_HOME: "/var/www"
12     SSH_USER_PASSWORD_HASHED: "true"
13     SSH_USER_PASSWORD: "wordpress"
14     SSH_AUTHORIZED_KEYS: "ssh-rsa
15         AAAAB3NzaC1yc2EAAAADAQABAAQ9C9w9fZfn9JicyTq//
16         peCAd9gQ7mi4vCioBzx0zGXmSWsY+4TijbDK2Xb9ZuQw8tyCg9rjZ24W2mvfM6
17         +tpC7nVZRvvsS0ji641hN9FamBt34+
18         oTe0MXyE1WH5dGDdwLHYXAO/R/
19         yNB1xzT1QaqRjA9PJgcrV2LsnT/6rbrX1x3mOGOM1Kyt0THiHhU1NfRCnGFR
20         6Velkx8RNj9z8zI9kqaJTICew1Ss1WFD0+02
21         Ij26Xp4JdK4qCCRGsYv6QTbNekN7
22         icp25dYK1XsxTiT+N7CYvg0EEeb/
23         lRRzYX9c0JWbaqhiASLo1cYwkWCONqewjID40QkYjN9JrqdOnb/"
24     restart: "always"

```

Con esto tendríamos la conexión SSH asegurada, recibiendo el estado **OK** en la GUI de Naemon.

5.4.2.1.7. Servicio para comprobar el protocolo HTTP A través de la directiva **check_command** haremos uso del plugin **check_http**, con este plugin comprobaremos el servicio HTTP en el host especificado. Se puede probar servidores normales (**http**) y seguros (**https**). El uso de este comando suele ser de la siguiente forma:

```

1  check_http -H <vhost> | -I <IP-address> [-u <uri>] [-p <port>]
2  [-J <client certificate file>] [-K <private key>]
3  [-w <warn time>] [-c <critical time>] [-t <timeout>] [-L] [-E] [-a
4  auth]
5  [-b proxy_auth] [-f <ok|warning|critical|follow|sticky|stickyport>]
6  [-e <expect>] [-d string] [-s string] [-l] [-r <regex> | -R <case-
7  insensitive regex>]
8  [-P string] [-m <min_pg_size>:<max_pg_size>] [-4|-6] [-N] [-M <age>]
9  [-A string] [-k string] [-S <version>] [--sni] [-C <warn_age>[,<
10 crit_age>]]
11 [-T <content-type>] [-j method]

```

La función de **check_command** aplicará dicho comando plugin con la siguiente forma de definición:

```
1 check_http!-u /naemon/
```

Donde se asume que:

- La ruta asignada lo realizará a través de -u asignando la ruta /naemon/

Si este comando se accediera a través de una terminal o línea de comandos de forma expandida, se escribiría de la siguiente forma:

```
1 /usr/lib/naemon/plugins/check_http -I 127.0.0.1 -u /naemon/
```

Si accedemos a la **GUI de Thruk** podemos observar el registro de información del rendimiento, obteniéndose los siguientes valores:

```
1 HTTP OK: HTTP/1.1 302 Found - 502 bytes in 0.001 second response time
2
3 time=0.000893s;;;0.000000;10.000000 size=502B;;;0
```

Por lo que el estado que obtendremos será **OK**, especificando que se ha encontrado el protocolo y el tiempo de respuesta es considerable para su acceso, es decir, realiza una petición GET o POST a dicha ruta asignada.

5.5. Puesta en práctica: creación de complemento o plugin

A continuación para empezar a realizar pruebas crearemos un plugin capaz de comprobar si existe una página concreta, esto es en el caso de que el archivo **wp-config.php** no se detecte, ya que si no se realiza esta comprobación puede ser que tengamos cuando se realiza el servicio HTTP un OK, cuando este no es el verdadero resultado. Así además se comprueba que la base de datos funciona correctamente. Para comprobar que accede de forma correcta a la base de datos y recoge de forma correcta la información haremos uso del plugin **check_http**, aunque realizaremos una modificación, por lo que crearemos un nuevo comando, llamado **check_wp_dominio** y le especificaremos que acceda a la ruta **/2019/08/16/hola-mundo/**, esta se trata de un artículo concreto de nuestro sistema WordPress.

Después para poder hacer uso y mostrar el resultado, definiremos el servicio, donde sólo llamaremos al comando creado heredando como en los anteriores servicios el servicio **local-service**.

```

1 define command{
2     command_name      check_wp_dominio
3     command_line      $USER1$/check_http -H $HOSTADDRESS$ -u
                        /2019/08/16/hola-mundo/
4 }
5 define service {
6     service_description WordPress Hola Mundo
7     host_name          wordpress
8     use                local-service
9     check_command       check_wp_dominio
10 }

```

En la figura 5.14 se muestra el resultado obtenido por el servicio ejecutado anteriormente definido.

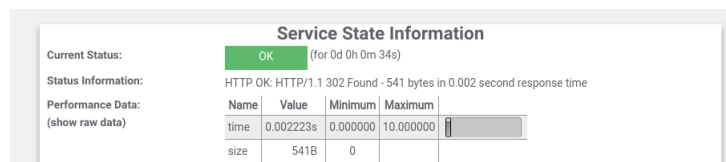


Figura 5.14: Información del estado del plugin de comprobación del puerto 80 sobre la página del artículo Hola, mundo de WordPress

Ahora para dar funcionamiento a nuestro sistema y además aplicar la creación de un nuevo plugin o complemento, crearemos uno por el cual compruebe que la ruta indicada por nuestro servicio no se encuentre infectada, es decir, al encontrarnos ante un sistema CMS como WordPress estamos expuestos a gran cantidad de vulnerabilidades externas, por lo que nos apoyaremos de Naemon para monitorizar este estado, para ello crearemos el plugin con nombre **check_hackedfiles**.

El contenido de este plugin es el siguiente:

```

1
2 #!/bin/bash
3 #
4
5 # Config regular expression
6 REG_EXPR[1]="[0-9a-zA-Z][0-9a-zA-Z][0-9a-zA-Z][0-9a-zA-Z][0-9a-zA-Z]
    [0-9a-zA-Z][0-9a-zA-Z][0-9a-zA-Z][0-9a-zA-Z][0-9a-zA-Z][0-9a-zA-Z]
    [0-9a-zA-Z][0-9a-zA-Z][0-9a-zA-Z][0-9a-zA-Z][0-9a-zA-Z][0-9a-zA-Z]
    [0-9a-zA-Z][0-9a-zA-Z][0-9a-zA-Z][0-9a-zA-Z][0-9a-zA-Z][0-9a-zA-Z]"
7 REG_EXPR[2]="[\\]x[0-9a-zA-Z][0-9a-zA-Z][\\]x[0-9a-zA-Z][0-9a-zA-Z][\\]x
    [0-9a-zA-Z][0-9a-zA-Z][\\]x[0-9a-zA-Z][0-9a-zA-Z]"
8 REG_EXPR[3]="[\\]x[0-9a-zA-Z][0-9a-zA-Z][0-9a-zA-Z][\\]x[0-9a-zA-Z][0-9a
    -zA-Z][0-9a-zA-Z]"
9 REG_EXPR[4]="[\\]x[0-9a-zA-Z][0-9a-zA-Z][0-9a-zA-Z][0-9a-zA-Z]"
10 REG_EXPR[5]="_[00][00][00][00][00]"
11
12 # EO Config regular expression
13
14 # Collection of regular expressions
15 REG_EXPR=$(printf '%s|' "${REG_EXPR[@]}" | sed 's/|$/ /')
16
17 STATE_OK=0
18 STATE_WARNING=1
19 STATE_CRITICAL=2
20 STATE_UNKNOWN=3
21
22 USAGE="Usa: 'basename $0' [-h] [-d DIR] [-b NUM] [-l LOG] [-s w/c] [-x
    FILE]\n
23 -d [DIR] Directory to check.\n
24 -b [NUM] Days back.\n
25 -l [FILE] Log file.\n
26 -s [w]arning or [c]ritical: Select state when hacked files are present
    . Default: w\n
27 -x [FILE] Exclude. List of files to skip check.\n
28 -e [EXTENSIONS] Exclude file extensions. e.g. \"*.jpg|*.png\"\n
29 Please use full paths!
30 "
31
32 if [ "$1" == "-h" ] || [ "$1" == "" ] ; then
33 echo -e $USAGE
34 exit $STATE_UNKNOWN
35 fi
36
37 while getopts d:b:l:s:x:e: option
38 do
39 case "$option"
40 in
41 d) CHECK_DIR=${OPTARG};;
42 b) DAYS_BACK=${OPTARG};;
43 l) LOG_FILE=${OPTARG};;
44 s) RETURN_STATE=${OPTARG};;
45 x) EXCLUDE_FILES=${OPTARG};;
46 e) EXCLUDE_FILE_EXTENSIONS=${OPTARG};;
47 esac
48 done
49
50 ### Exit State
51 if [ -z $RETURN_STATE ]; then
52 STATE=$STATE_WARNING

```

```

53 else
54 if [ $RETURN_STATE == "w" ]; then
55 STATE=$STATE_WARNING
56 elif [ $RETURN_STATE == "c" ]; then
57 STATE=$STATE_CRITICAL
58 else
59 echo "Bad value for -s!"
60 echo -e $USAGE
61 exit $STATE_UNKNOWN
62 fi
63 fi
64 ### EO Exit State
65
66 ### Excludes
67 if [ -z $EXCLUDE_FILES ]; then
68 EXCLUDE_PART=""
69 else
70 if [ -s "$EXCLUDE_FILES" ]; then
71 EXCLUDE_PART=$(printf "! -samefile %s " $(cat $EXCLUDE_FILES))
72 else
73 EXCLUDE_PART=""
74 fi
75 fi
76 ### EO Excludes
77
78 ### Exclude file extensions
79 if [ -z $EXCLUDE_FILE_EXTENSIONS ]; then
80 EXCLUDE_EXTENSIONS_PART=""
81 else
82 EXCLUDE_EXTENSIONS_PART=$(printf "! -name %s " $(echo
83     $EXCLUDE_FILE_EXTENSIONS | sed 's/|/ /g'))
84 fi
85 ### EO Exclude file extensions
86
87 DATE=$(date)
88 CHECK=$(find $CHECK_DIR -type f \( -name "*" $EXCLUDE_EXTENSIONS_PART
89     $EXCLUDE_PART \) -ctime -$DAYS_BACK -exec grep -lrE "$REG_EXPR" {}
90     \;)
91
92 echo "----" >> $LOG_FILE
93 echo $DATE >> $LOG_FILE
94
95 printf '%s\n' "${CHECK[@]}" >> $LOG_FILE
96
97 if [ -z "$CHECK" ]; then
98 echo "OK - There are no infected files for the last $DAYS_BACK days."
99 exit $STATE_OK
100 else
101 echo "There may be infected files from the last $DAYS_BACK days."
102 exit $STATE
103 fi

```


Su **funcionamiento** es el siguiente:

- Dependiendo el valor o la opción a comprobar que se le pase que se le pase, ya sea un directorio, un archivo log, un archivo excluido o un archivo de extensión, parseará con **getopts** esa opción y la guardará en su correspondiente opción.
- Si se ha encontrado con un algún estado **WARNING O CRITICAL** se encargará de devolver dicho estado.
- Empieza la comprobación de cada elementos comprobando que se encuentra dentro del sistema.
- En el caso de encontrarse y además contener expresiones regulares que no se encuentran acorde al contenido que deben tener, esta lanzará la opción de que se encuentran infectados y lanzará un **WARNING o CRITICAL**.

Para poder poner este plugin en marcha, es necesario crear el comando correspondiente y su servicio.

```
1
2 define command{
3     command_name      check_hackedfiles
4     command_line      $USER1$/check_hackedfiles $ARG1$
5 }
6
7 define service{
8     use                generic-service,service-pnp
9     host_name          wordpress
10    service_description Hackedfiles
11    check_command       check_hackedfiles! /
12 }
```

Este contenido lo ingresaremos en **wordpresshosts.cfg** y **wordpressservices.cfg** anteriormente creados y además debemos crear dentro de nuestro **Dockerfile** la siguiente línea para poder añadir a la carpeta de plugin nuestro archivo nuevo, y le daremos permiso de ejecución:

```
1 ADD plugins/check_hackedfiles /usr/lib/naemon/plugins/
   check_hackedfiles
2 RUN sync && chmod 755 /usr/lib/naemon/plugins/check_hackedfiles
```

Si lanzamos nuestra ejecución y accedemos a nuestro servicio creado en **Thruk** podemos visualizar lo reflejado en la figura 5.15.

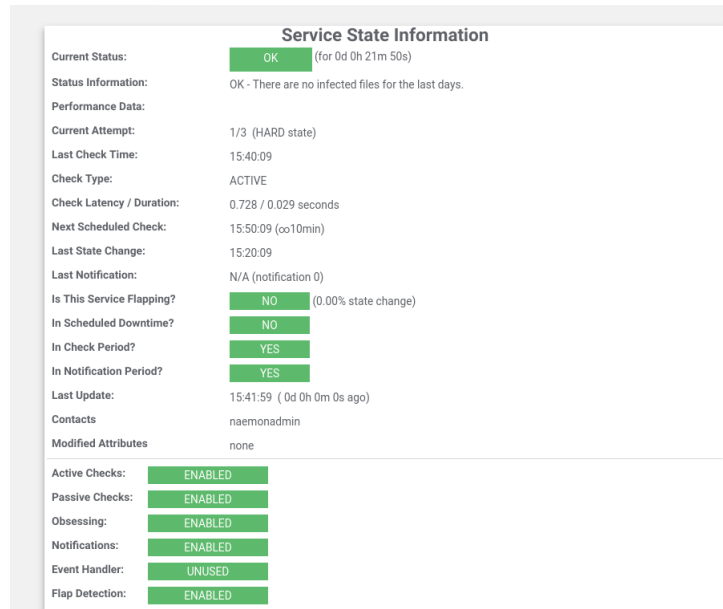


Figura 5.15: Información del estado del plugin de comprobación de archivos

En el capítulo siguiente pasamos a realizar el **modelado de cargas** de todos los servicios creados en **Naemon**, es decir, representaremos la carga de trabajo mediante el análisis de prestaciones y crear con esto un modelo.

Capítulo 6

Modelado

6.1. Introducción a la carga de trabajo

[32] A la hora de realizar un análisis de prestaciones siempre se tiene que tener en cuenta el comportamiento que va a tener un sistema o varios ante una carga concreta de trabajo.

Lo normal es que no se disponga en tiempo real de esta carga a la que se va someter el sistema, pero sí se puede utilizar o aplicar un modelo con características similares o parecidas. Por lo que estos análisis se suelen basar en **modelos de carga**, es decir, modelos que se extraen tras la caracterización previa de la propia carga. Cuando el modelo se encuentra disponible, se pasa a estudiar los efectos o cambios en la carga y en el sistema, cambiando parámetros.

En todo momento se ha hablado de **carga de trabajo**, esta se trata del conjunto de todas las peticiones que el sistema recibe de su entorno durante un periodo de tiempo dado.

El **análisis de la carga** es un papel fundamental en cualquier estudio en los que hay que determinar **índices de rendimiento**, estos se encuentran directamente relacionados con la carga y no se pueden expresar de forma independiente a ésta. Además el índice de rendimiento siempre debe ir determinado de la información de la carga bajo la que fue determinado.

El modelo de carga ha de capturar el comportamiento estático y dinámico de la carga real y ha de ser compacto, repetible y preciso. Este modelo de carga supone una descripción cuantitativa de las características de la carga, a esta descripción se le denomina **caracterización de la carga**.

Se llama **caracterización de la carga** al proceso por el cual se define un modelo de carga que reproduce lo mejor posible las características de la carga real.

El modelo ha de establecerse en función de los parámetros que pueden afectar al comportamiento del sistema. Por lo que una **carga** está perfectamente caracterizada si su resultado es un conjunto de parámetros cuantitativos seleccionados de acuerdo con los objetivos de la operación de caracterización.

6.1.1. Características de un modelo de carga

Un modelo de carga debe contar con las siguientes características:

- **Reproducibilidad:** deben ser capaces de reproducir la carga de prueba sobre todo en situaciones de ajuste de sistemas o de comparación de sistemas.
- **Representatividad:** los aspectos de la carga real han de estar representados en el modelo.
- **Compacidad:** se recomienda usar modelos de carga compactos que permitan realizar las mediciones del sistema en tiempos cortos.
- **Privacidad:** el uso de modelos nos permite evitar problemas de privacidad y seguridad.
- **Consistencia/Coherencia:** se necesita contar con representación de la carga consistente con la aplicación.
- **Flexibilidad:** posibilidad de variar los parámetros del modelo de carga para ajustarlo a las variaciones que se produzcan en el sistema real.
- **Independencia del sistema:** no debe variar el sistema sobre el que se procesa.

Los pasos principales para su construcción son los siguientes:

- **Formulación:** seleccionar los parámetros para la descripción de la carga y definir el criterio de evaluación de la representatividad del modelo.
- **Recolección de parámetros de la carga**
- **Análisis estadístico de los datos medidos:** podemos realizar varios tipos:
 - **Análisis preliminar:** partición de los datos.
 - **Análisis de las distribuciones de los parámetros**
 - **Muestreo:** si la cantidad es muy grande, se elige una muestra de los datos medidos para conseguir un tiempo de procesamiento y una cantidad de espacio de almacenamiento razonable.
 - **Análisis estático:** clasificación y partición de los componentes de la carga, diferenciamos entre análisis de componentes y análisis cluster.
 - **Análisis dinámico:** se realiza cuando hay que reproducir las características de variación temporal de la carga.
- **Representatividad:** en esta parte verificamos el modelo a través de:
 - **Consistencia de los componentes del modelo**
 - **Consistencia con el sistema**
 - **Consistencia con la carga real**

Para ello planteamos lo siguiente: *Un modelo de carga W' es perfectamente representativo de W si produce los mismos valores de los índices de rendimiento P que son obtenidos cuando W se ejecuta sobre el sistema S .*

6.1.2. Selección de la carga de trabajo

Hay que tener en cuenta que si la carga de trabajo sobre la que se realiza el estudio de rendimiento no se selecciona de forma correcta, puede llegar a producir conclusiones incorrectas.

6.2. Obtención de datos de rendimiento a través de PNP4Nagios

Por lo que es **importante** tener en cuenta lo siguiente:

- Los servicios que se prestan
- El nivel de detalle
- La representatividad
- El impacto de componentes externos
- Realización de repeticiones

A la hora de seleccionar el servicio tenemos que tener en cuenta en la situación que nos encontremos en este caso el servicio será para transacciones **HTTP** y **PING** de nuestra aplicación o sistema WordPress.

Es importante analizar el punto de vista de la naturaleza del servicio prestado identificando la forma en la que se encuentra, es decir, el entorno concreto, en este caso se lanzarán peticiones **GET** y **POST** hacia el sistema **WordPress**.

Para realizar dicho análisis debemos obtener los diferentes datos de rendimiento y poder generar unas gráficas, a continuación explicaremos como realizamos esto a través de un complemento que ofrece Naemon.

6.2. Obtención de datos de rendimiento a través de PNP4Nagios

Para realizar las distintas pruebas de rendimiento es necesario generar gráficas y hacer recogida de datos para ello haremos uso de la herramienta **PNP4Nagios**.

6.2.1. ¿Qué es PNP4NAGIOS?

PNP4Nagios [33] es un modulo para **Naemon** y además para **Nagios** que analiza los datos de rendimiento de los servicios que tengamos implementados en cada host, almacena automáticamente los datos en bases de datos RRD (bases de datos Round Robin).



Figura 6.1: PNP4Nagios

Te muestra en forma de gráficos cada servicio de cada host de diferentes periodos de tiempo, además **PNP4Nagios** es un módulo oportuno para los administradores de redes, ya que tienes un buen control administrativo de todos los servicios de diferentes periodos de tiempo, pudiendo hacer comparativas de calidad entre los mismos servicios en diferentes periodos de tiempo entre otras opciones que desee el administrador.

A la hora de realizar la instalación de **PNP4Nagios** es necesario la incorporación del componente **NPCD(Nagios-Perfdata-C-Daemon)** que nos permite en forma asíncrona disponer o manejar los datos de rendimiento de Naemon.

NPCD ofrece como ventajas la mejora del rendimiento para Naemon debido a que el procesamiento de datos del rendimiento se separa del núcleo de Naemon, teniendo así más tiempo para el trabajo de éste. Además no se pierden datos, siempre que se escriban los archivos perfdata en el directorio **spool**, esto es un directorio que contiene archivos a la espera de ser imprimidos hasta que el dispositivo esté listo, aunque NPCD muera o se olvide reiniciar el sistema.

Cuenta con la desventaja que la información de rendimiento no es en tiempo real, ya que existe un retraso en la estructura de los archivos de datos de rendimiento de Naemon(`service_perfdata_file_processing_interval`), además existe otro retraso dentro de **NPCD** que espera hasta 10 segundos después de cada exploración del directorio.

6.2. Obtención de datos de rendimiento a través de PNP4Nagios

6.2.2. Instalación de PNP4Nagios a través de Docker

Para poder realizar la instalación de **PNP4Nagios** utilizaremos el archivo **Dockerfile** creado en 3 para ello debemos empezar con la instalación de las dependencias, esto lo haremos de la siguiente forma:

```
1      RUN apt-get update && \  
2      DEBIAN_FRONTEND=noninteractive \  
3      apt-get install -y \  
4      make \  
5      rrdtool \  
6      librrds-perl \  
7      g++ \  
8      php-cli \  
9      php-gd \  
10     php-xml \  
11     libapache2-mod-php
```

Una vez instaladas las dependencias es necesario descargar PNP4Nagios a través del siguiente enlace <http://downloads.sourceforge.net/project/pnp4nagios/PNP-0.6/pnp4nagios-0.6.24.tar.gz> y extraer su contenido, esto quedará reflejado en el Dockerfile de la siguiente forma:

```
1      RUN wget -O pnp4nagios.zip https://github.com/linge/\  
2      pnp4nagios/archive/master.zip && \  
3      unzip pnp4nagios.zip && \  
4      cd pnp4nagios-master && \  
5      ./configure --with-nagios-user=naemon --with-nagios-group=\  
6      naemon && \  
7      make all && \  
8      make install && \  
9      make install-webconf && \  
10     make install-config && \  
11     make install-init && \  
12     cd ../ && \  
13     rm -rf pnp4nagios.zip pnp4nagios-master
```

Lo que estamos haciendo es descargar el archivo zip, extraerlo y realizar la construcción e instalación de **PNP4Nagios**.

A continuación pasaremos a explicar la forma de integrar PNP4Nagios en Naemon modificando una serie de archivos.

6.2.2.1. Integración de PNP4Nagios en Naemon

Para empezar a integrar en perfectas condiciones PNP4Nagios en Naemon es necesario modificar su archivo **pnp4nagios.cfg**, como nos encontramos en Ubuntu debemos mover el archivo desde la carpeta de configuración de **httpd** a la de Apache y luego establecer un enlace simbólico de **conf-available** a **conf-enabled** (esto es debido a las nuevas modificaciones de Apache en Ubuntu).

También debemos modificar el archivo **config.local.php**, cambiando la configuración de **nagios base**, estableciendo la que va a ser utilizada en este caso **/thruk/cgi-bin**.

Además debemos de modificar unas líneas concretas de este archivo y sustituir Nagios por Naemon, es decir, esto en el Dockerfile se verá reflejado de la siguiente manera:

```
1 RUN mv /etc/httpd/conf.d/pnp4nagios.conf /etc/apache2/conf-available
   && \
2 ln -sf /etc/apache2/conf-available/pnp4nagios.conf /etc/apache2/conf-
   enabled/pnp4nagios.conf && \
3 sed -i "s|\\$conf\\['nagios_base'\\].*=.*\\.\\.\\\";|\\$conf['nagios_base'] =
   \\\"/thruk/cgi-bin\\\";|" /usr/local/pnp4nagios/etc/config_local.php
```

Una vez instalado nos dirigimos a configurar el modo que vamos a usar en el **PNP4Nagios**, en mi caso he usado el modo síncrono es la forma más fácil de integrar en Naemon el recolector de datos “process_perfddata.pl”.

Inicialmente hemos debido habilitar el procesamiento de los datos de rendimiento en **/etc/naemon/naemon.cfg**, esta directiva ya estará presente en el fichero de configuración y el valor por defecto es “0”. Lo cambiaremos a 1, es decir, cambiaremos ese valor de la siguiente forma en el archivo **run.bash**

```
1 if grep -q 'process_performance_data=0' /data/etc/naemon/naemon.cfg;
   then
2
3 echo "Started PNP4Nagios setup"
4 sed -i 's|process_performance_data=0|process_performance_data=1|' /
   data/etc/naemon/naemon.cfg
```

De forma seguida debemos añadir las siguientes entradas en el mismo archivo **naemon.cfg** para configurar los ajustes de rendimiento de la información. Además debemos especificar el comando para procesar los datos de rendimiento, para así después crear los comandos referenciados, además de la opción de poder hacer uso de **PNP4Nagios** en el host y los servicios.

12. Obtención de datos de rendimiento a través de PNP4Nagios

```
1 if grep -q 'process_performance_data=0' /data/etc/naemon/naemon.cfg;
  then
2
3 echo "Started PNP4Nagios setup"
4 sed -i 's|process_performance_data=0|process_performance_data=1|' /
  data/etc/naemon/naemon.cfg
5
6 cat <<'EOT' >> /data/etc/naemon/naemon.cfg
7 #
8 # service performance data
9 #
10 service_perfdata_file=/usr/local/pnp4nagios/var/service-perfdata
11 service_perfdata_file_template=DATATYPE::SERVICEPERFDATA\tTIMET::
  $TIMET$\tHOSTNAME::$HOSTNAME$\tSERVICEDESC::$SERVICEDESC$\
  tSERVICEPERFDATA::$SERVICEPERFDATA$\tSERVICECHECKCOMMAND::
  $SERVICECHECKCOMMAND$\tHOSTSTATE::$HOSTSTATE$\tHOSTSTATETYPE::
  $HOSTSTATETYPE$\tSERVICESTATE::$SERVICESTATE$\tSERVICESTATETYPE::
  $SERVICESTATETYPE$
12 service_perfdata_file_mode=a
13 service_perfdata_file_processing_interval=15
14 service_perfdata_file_processing_command=process-service-perfdata-file
15 #
16 #
17 #
18 host_perfdata_file=/usr/local/pnp4nagios/var/host-perfdata
19 host_perfdata_file_template=DATATYPE::HOSTPERFDATA\tTIMET::$TIMET$\
  tHOSTNAME::$HOSTNAME$\tHOSTPERFDATA::$HOSTPERFDATA$\
  tHOSTCHECKCOMMAND::$HOSTCHECKCOMMAND$\tHOSTSTATE::$HOSTSTATE$\
  tHOSTSTATETYPE::$HOSTSTATETYPE$
20 host_perfdata_file_mode=a
21 host_perfdata_file_processing_interval=15
22 host_perfdata_file_processing_command=process-host-perfdata-file
23 EOT
24 echo "Started include PNP4NAGIOS in commands.cfg"
25 cat <<'EOT' > /data/etc/naemon/conf.d/pnp4nagios_commands.cfg
26 define command{
27     command_name    process-service-perfdata-file
28     command_line    /bin/mv /usr/local/pnp4nagios/var/service-
      perfdata /usr/local/pnp4nagios/var/spool/service-perfdata.
      $TIMET$
29 }
30 define command{
31     command_name    process-host-perfdata-file
32     command_line    /bin/mv /usr/local/pnp4nagios/var/host-
      perfdata /usr/local/pnp4nagios/var/spool/host-perfdata.
      $TIMET$
33 }
34 EOT
35 echo "Started include PNP4NAGIOS in hosts.cfg"
36 cat <<'EOT' >> /data/etc/naemon/conf.d/templates/hosts.cfg
37 define host {
38     name host-pnp
39     process_perf_data 1
40     action_url /pnp4nagios/index.php/graph?host=$HOSTNAME&$srv=
      _HOST_ ' class='tips' rel='/pnp4nagios/index.php/popup?host
      =$HOSTNAME&$srv=_HOST_
41     register 0
42 }
43 EOT
44 echo "Started include PNP4NAGIOS in commands.cfg"
45 cat <<'EOT' >> /data/etc/naemon/conf.d/templates/services.cfg
```

```
46 define service {
47     name service-pnp
48     process_perf_data 1
49     action_url /pnp4nagios/index.php/graph?host=$HOSTNAME$&srv=
        $SERVICEDESC$ ' class='tips' rel='/pnp4nagios/index.php/
        popup?host=$HOSTNAME$&srv=$SERVICEDESC$
50     register 0
51 }
52 EOT
53 fi
```

Sólo nos quedará activar el **servicio NPCD** para ello en **run.bash** hacemos uso de la siguiente llamada:

```
1 service npcd start
```

Sólo quedará borrar el archivo de instalación para que no hay ningún problema en la modificación.

```
1 RUN rm -f /usr/local/pnp4nagios/share/install.php
```

Si queremos probar cualquier servicio o host sólo tenemos que agregar en use la opción de **host-pnp** o **service-pnp**. Si accedemos a la dirección <http://localhost:3/pnp4nagios/> no podremos acceder a ella ya que nos faltaría realizar la integración con **Thruk**, que será lo siguiente a explicar.

6.2.2.2. Integración de PNP4Nagios en Thruk

Los **gráficos PNP4Nagios** se pueden integrar fácilmente en **Thruk**, lo que los hará aparecer en el servicio o en los detalles del host.

La integración se realiza especificando un **'action_url'** con la ruta a la instalación de **pnp4nagios**.

Primero es necesario la **autenticación de las cookies**, para ello debemos copiar el archivo de inclusión de autenticación de cookies (generalmente ubicado en **'/usr/share/thruk/thruk_cookie_auth.include'**) a su directorio de configuración httpd y modificar cada vez que ocurra la siguiente declaración RewriteCond desde:

```
1 RUN sed -i 's;RewriteCond\s\+%{REQUEST_URI}\s\+~\s/thruk$;RewriteCond
    %{REQUEST_URI} ~/(thruk|pnp4nagios);g' /usr/share/thruk/
    truk_cookie_auth.include && \
```

12.2. Obtención de datos de rendimiento a través de PNP4Nagios

Además muy importante es agregar la ruta utilizada para las cookies a la configuración local de **Thruk**, es decir, agregando la siguiente línea a `/etc/thruk/thruk.conf`.

```
1 cookie_path = /
```

Esto quedará reflejado de la siguiente forma en el archivo `run.bash`.

```
1 if grep -q "#cookie_path" /data/etc/thruk/thruk.conf; then
2 echo "Cambio de cookie_path"
3 sed -i 's|#cookie_path|cookie_path|' /data/etc/thruk/thruk.conf
4 fi
```

Una tengamos todo esto configurado e instalado accedemos a `http://localhost:3/pnp4nagios/` donde nos aparecerán los gráficos generados de cada servicio vinculado con PNP4Nagios, además de la gráfica del host vinculado. Todo esto se ve reflejado en la figura 6.2.

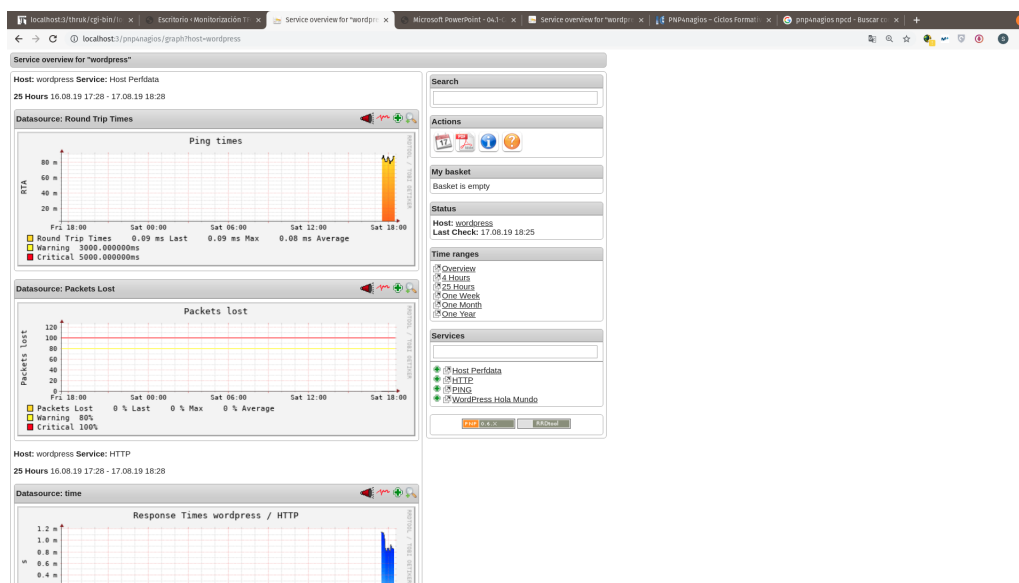


Figura 6.2: Estructura de PNP4Nagios

6.2.3. Exportación de los datos a formato CSV

Como ya tenemos todo para poder generar los datos, vamos a hacer uso de la herramienta que incluye PNP4Nagios para exportar en CSV, estos datos generados son los correspondientes a los datos generados por los servicios creados en el capítulo 5, donde realizábamos el servicio para comprobar **HTTP** y **PING**. Para ello la API [34] nos ofrece la forma de exportación a CSV de la siguiente forma: `/pnp4nagios/xport/<format>?host=<hostname>&srv=<servicedesc>`. Donde:

- *< format >* especifica el formato de exportación, teniéndose la opción de XML, JSON y CSV.
- *< hostname >* especificaremos el nombre del host, en este caso introducimos el nombre **wordpress**.
- *< servicedesc >* especificaremos el servicio que queremos exportar su información.

Como queremos obtener la información de rendimiento del análisis de **HTTP** y **PING** solo tendremos que acceder a las siguientes direcciones: `http://localhost:3/pnp4nagios/xport/CSV?host=wordpress&srv=PING` `http://localhost:3/pnp4nagios/xport/CSV?host=wordpress&srv=HTTP` Al acceder a estas dos direcciones obtendremos las tablas 8.1 y 8.2 encontradas en el Anexo 8. Esta tabla se ha realizado con la toma de datos durante treinta minutos, durante este tiempo el sistema se encarga de mandar peticiones de tipo **HTTP** a la ruta `/`, obteniendo su tiempo de respuesta y el tamaño de los paquetes procesados. En el caso de la realización de las pruebas para el servicio **PING** comprueba durante treinta minutos la pérdida de paquetes y el tiempo RTA generado cuando se realiza una llamada al comando ping al host wordpress. Esto queda quedará generado de la siguiente forma con el complemento **PNP4Nagios** en las figuras 6.3 y 6.4.

130. Obtención de datos de rendimiento a través de PNP4Nagios

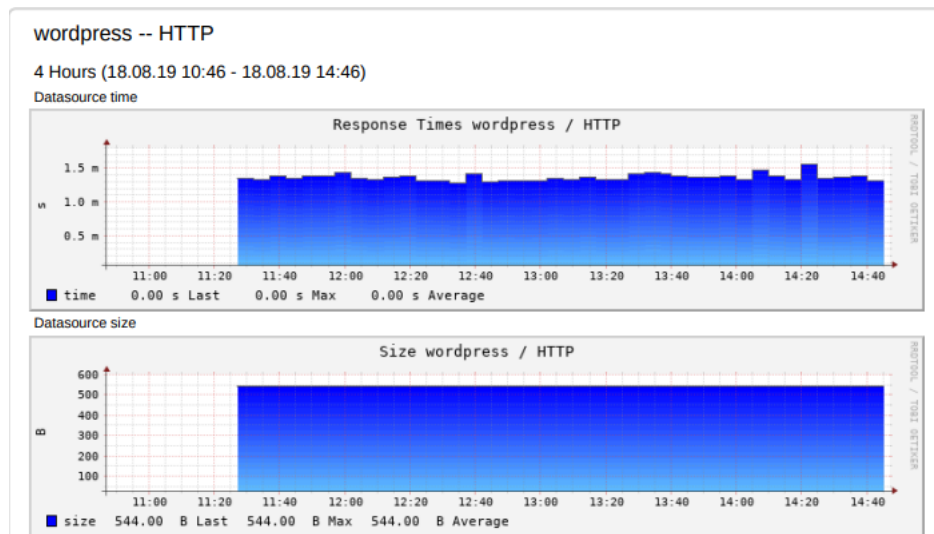


Figura 6.3: HTTP en interfaz PNP4Nagios

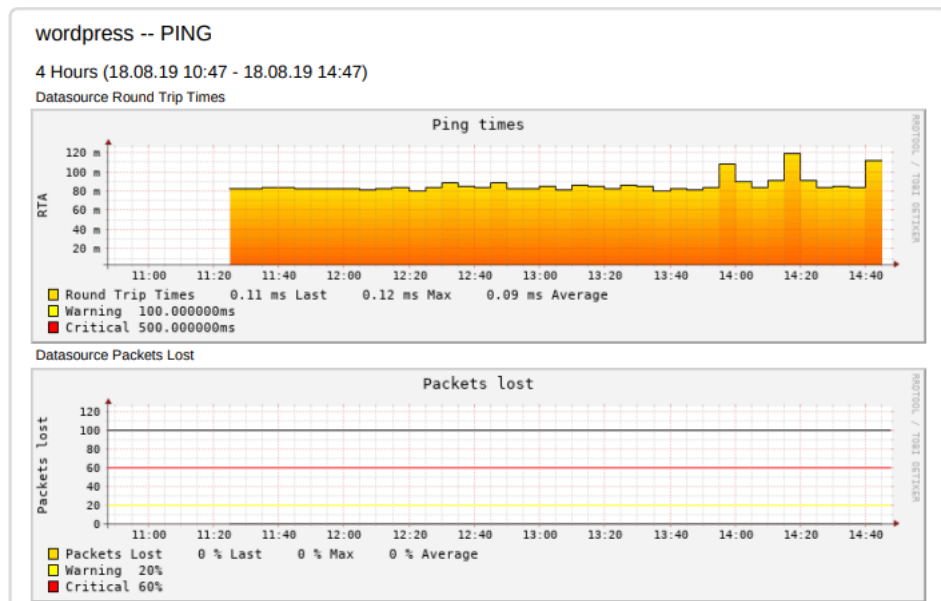


Figura 6.4: PING en interfaz PNP4Nagios

Si pasamos a la generación de gráficos con los resultados obtenidos podemos obtener los siguientes gráficos:

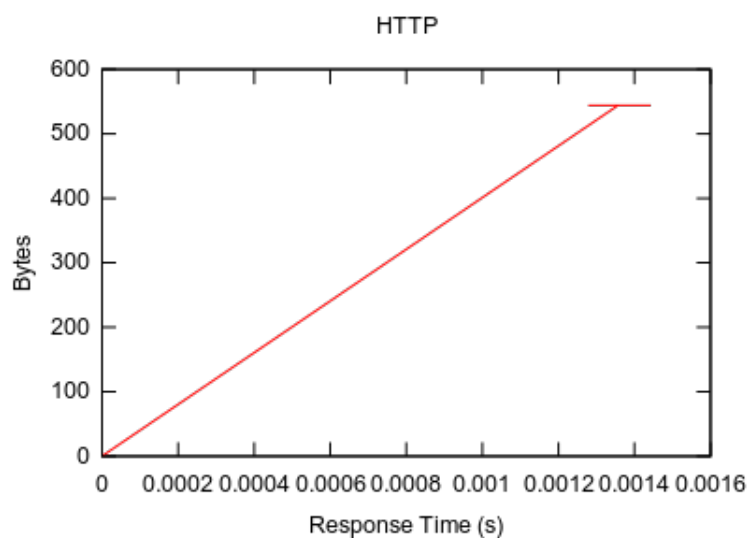


Figura 6.5: Gráfica del servicio HTTP

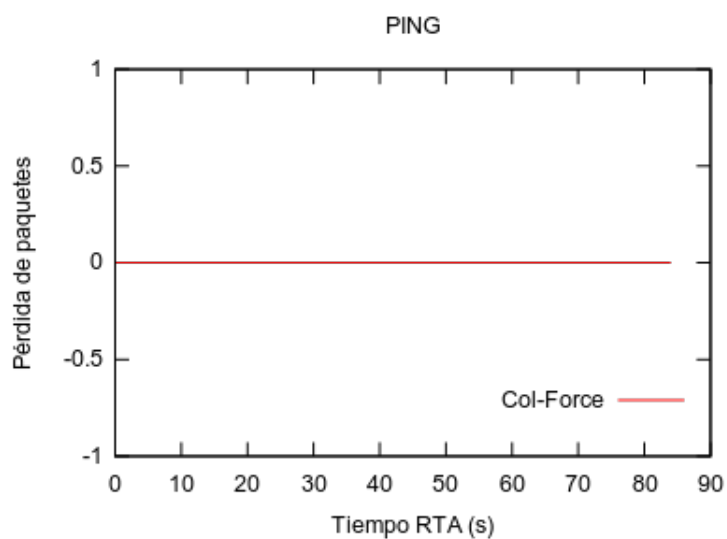


Figura 6.6: Gráfica del servicio PING

En el capítulo 7 se realizará una conclusión con los resultados obtenidos en el análisis de rendimiento de nuestro sistema.

Capítulo 7

Conclusiones y trabajos futuros.

En este capítulo se va a hablar de las conclusiones que se pueden observar a partir de los resultados obtenidos en el capítulo anterior y trabajos futuros que se desearían realizar a partir de lo ya concluido con este trabajo.

7.1. Conclusión.

Con este trabajo se ha querido realizar el despliegue completo de la herramienta de monitorización **Naemon** utilizando para el despliegue el uso de contenedores Docker, además de apoyarnos con la **interfaz GUI Thruk** para mostrar los resultados de dicha herramienta de forma visual.

Además se añadió la creación de un **sistema CMS WordPress**, el cual sería el que analizaríamos en profundidad mediante la creación de servicios par medir el rendimiento de este sistema.

Pero antes de todo realizaríamos pruebas de carga en el sistema WordPress mediante el uso del framework de prueba de carga **Locust**.

Finalmente con todo el despliegue realizado mediante **Docker-Compose**, se realizó pruebas al sistema WordPress, mediante la creación de plugins y además para finalizar análisis del rendimiento de los servicios mediante la representación gráfica apoyándose de la herramienta complemento **PNP4Nagios**.

En cuanto a los resultados obtenidos en dicho análisis se aprecia como el sistema responde de forma positiva durante los treinta minutos de comprobación, ya que no pierde paquetes a la hora de realizar el **PING**, aplicando **tiempos RTA** bastante reducidos, además a la hora de mandar peticiones **HTTP**, éste responde de forma favorable puesto que los tiempos de respuesta son lo suficientemente pequeños para que no haya problemas de pérdida de conexión, además el tamaño de los paquetes generados son siempre los mismos por lo que no tendremos ninguna desfragmentación generada.

7.2. Trabajos futuros.

En cuanto a los trabajos futuros a partir del actual, el principal sería la realización de forma automatizada del despliegue pudiendo apoyarnos de la herramienta **Ansible**. Otra idea futura sería la adaptación de la pila ELK(Elasticsearch, Logstash y Kibana) de Elastic para recoger todos los registros generados durante la monitorización, haciendo que la búsqueda, análisis y visualización de los datos aparezcan con mayor facilidad en los dashboard, además de poder manejarse gran cantidad de datos de forma eficiente.

Capítulo 8

Anexo

Response TIME (s)	Size (Bytes)
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0.00135716	544
0.00135716	544
0.00135716	544
0.00135716	544
0.00135716	544
0.00133696	544
0.00133696	544
0.00133696	544
0.00133696	544
0.00133696	544
0.00139122	544
0.00139122	544
0.00139122	544
0.00139122	544
0.00139122	544
0.00134988	544
0.00134988	544
0.00134988	544
0.00134988	544
0.00134988	544
0.00138966	544
0.00138966	544
0.00138966	544
0.00138966	544
0.00138966	544
0.00138788	544
0.00138788	544
0.00138788	544
0.00138788	544
0.00138788	544
0.00144322	544
0.00144322	544
0.00144322	544
0.00144322	544

Tabla 8.1: Tabla HTTP

Tiempo RTA (s)	Porcentaje pérdida de paquetes
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
83	0
83	0
83	0
83	0
83	0
83	0
83	0
83	0
83	0
83	0
83	0
83	0
0.083876666667	0
0.083876666667	0
0.083876666667	0
0.083876666667	0
0.083876666667	0
84	0
84	0
84	0
84	0
84	0
0.082246666667	0
0.082246666667	0
0.082246666667	0
0.082246666667	0
0.082246666667	0
0.082876666667	0
0.082876666667	0
0.082876666667	0
0.082876666667	0
0.082876666667	0
0.082123333333	0
0.082123333333	0
0.082123333333	0
0.082123333333	0
0.082123333333	0

Tabla 8.2: Tabla PING

Bibliografía

- [1] *SNMP Protocol*. URL: <http://www.snmp.com/protocol/>.
- [2] URL: <https://mum.mikrotik.com/presentations/B014/freddy.pdf>.
- [3] *Zabbix*. URL: <https://www.zabbix.com/>.
- [4] URL: <https://xmpp.org/about/>.
- [5] URL: https://www.ibm.com/support/knowledgecenter/en/SSYKE2_7.1.0/com.ibm.java.security.component.71.doc/security-component/jsse2Docs/ssloverview.html.
- [6] *Cacti*. URL: <https://www.cacti.net/>.
- [7] URL: <https://www.ssh.com/ssh/protocol/>.
- [8] *Pandora FMS*. URL: <http://pandorafms.org/es/>.
- [9] *Nagios*. URL: <https://www.nagios.org/>.
- [10] URL: <https://diego.com.es/concepto-y-funcionamiento-de-cgi>.
- [11] *Naemon*. URL: <http://www.naemon.org/>.
- [12] URL: <https://www.naemon.org/documentation/usersguide/configmain.html>.
- [13] URL: <https://www.naemon.org/documentation/usersguide/config.html>.
- [14] Jawwad Shamsi y Monica Brocmeyer. "PRINCIPLES OF NETWORK MONITORING". En: (). URL: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.149.7296&rep=rep1&type=pdf>.
- [15] *Thruk*. URL: <https://www.thruk.org/index.html>.
- [16] URL: <https://www.naemon.org/documentation/usersguide/livestatus.html>.
- [17] URL: <https://www.naemon.org/documentation/usersguide/plugins.html>.

- [18] *Contenedores docker*. URL: <https://www.docker.com/>.
- [19] URL: <https://docs.docker.com/engine/>.
- [20] URL: <https://www.openshift.com/>.
- [21] URL: <https://coreos.com/rkt/docs/latest/>.
- [22] URL: <https://linuxcontainers.org/>.
- [23] URL: [https://wiki.archlinux.org/index.php/SysVinit_\(Espa%C3%B1ol\)](https://wiki.archlinux.org/index.php/SysVinit_(Espa%C3%B1ol)).
- [24] URL: <https://yaml.org/>.
- [25] Ian Molyneaux. “The Art of Application Performance Testing”. En: (). URL: <http://shop.oreilly.com/product/9780596520670.do>.
- [26] URL: <https://docs.microsoft.com/en-us/previous-versions/msp-n-p/bb924356%28v%3dpandp.10%29#performance-load-and-stress-testing>.
- [27] URL: <https://www.educba.com/jmeter-vs-gatling/>.
- [28] URL: <https://gatling.io/>.
- [29] *Load Testing Framework*. URL: <https://locust.io/>.
- [30] URL: <https://jmeter.apache.org/>.
- [31] *Writing a Locustfile*. URL: <https://docs.locust.io/en/stable/writing-a-locustfile.html>.
- [32] Xavier Molero Prieto. “Evaluación y modelado del rendimiento de los sistemas informáticos”. En: ().
- [33] URL: <https://docs.pnp4nagios.org/start>.
- [34] URL: https://report.cs.rutgers.edu/pnp4nagios/docs/view/en_US/xport.

