Icinga Network Monitoring

Monitor complex and large environments across dispersed locations with Icinga

Viranch Mehta

Icinga Network Monitoring

Monitor complex and large environments across dispersed locations with Icinga

Viranch Mehta

Icinga Network Monitoring

Credits

Author
Viranch Mehta

Reviewers
Toni de la Fuente (Blyx)
Naoya Hashimoto
Michael Medin
Daniel Oetken

Acquisition Editor
Vinay Argekar

Lead Technical Editor
Amit Ghodake

Technical Editors
Sharvari H. Baet
Kanhucharan Panda

Copy Editors
Alisha Aranha
Roshni Banerjee
Lavina Pereira

Project Coordinator
Amigya Khurana

Proofreader
Bob Phillips

Indexer
Tejal R. Soni

Graphics
Yuvraj Mannari

Production Coordinator
Kyle Albuquerque

Cover Work
Kyle Albuquerque

About the Author

Viranch Mehta is fresh out of college, with experience of one year at Directi Internet Solutions (`www.directi.com`) as a Developer in Operations. He has done his B.Tech. (ICT) from Dhirubhai Ambani Institute of Information and Communication Technology (`www.daiict.ac.in`), Gandhinagar (Gujarat). He has been a student Developer for Google Summer of Code (`www.google-melange.com`) in 2011 and 2012 with the KDE (`www.kde.org`) project in college, and a mentor in 2013.

The author has been a Linux and system administration enthusiast since his college days. He has built a lot of Homebrew automation programs using bash/python scripting as daily tasks in college. He started developing extensively with the KDE project, working primarily with the Plasma subproject, and later with the KDE Games subproject. He has also worked on Qt (`qt-project.org`) applications and writes small-scale software using Qt.

Apart from the work/hobby projects in college, the author has spent a large amount of time as a professional, building and deploying tools for monitoring and alert management. Directi, being an Internet solutions (such as domain name registration, Windows/Linux hosting, contextual advertising, and so on) provider, has a large-scale server and network infrastructure. He has also briefly contributed to Icinga with a few bug fixes.

About the Reviewers

Toni de la Fuente (Blyx) is a Senior Solutions Engineer for Americas at Alfresco Software Inc. The highlight of his career is the more-than-14 years' experience he has in Systems Administration and Networking and Security. He also teaches LPI Linux certification, Red Hat Certified Engineer (RHCE), and ITIL v3; recently, he was certified as an AWS Technical Professional and AWS Business Professional.

He was declared an Open Source enthusiast, having founded different open source projects in the last few years. He has participated in other open source-related projects, such as Madrid Wireless, Fedora Linux, or OpenSolaris Hispano, and been referenced in books on network security. He regularly takes lectures, courses, and conferences at different events in Europe, the United States, and Latin America. He has also contributed to the world of Open Source for more than 10 years with his blog `http://blyx.com` and through Twitter (`@ToniBlyx`).

> Toni wants to thank Packt Publishing for their trust in him and to all the people who spend tons of hours working at night making Open Source—you all make this world a better place, keep going.

Naoya Hashimoto has been working on Linux system integration and system and operation maintenance, both on-premise and on a public cloud for years. He has also started developing a new service to manage, maintain, and monitor a system on public cloud along with his experience in working as an infrastructure engineer in Japan for the past few years. He has worked on books such as OSS DB Standard Text – PostgreSQL (`http://www.oss-db.jp/ossdbtext/text.shtml`) and was also involved in the translation of some chapters from Japanese to English.

Michael Medin is a Senior Developer and Lead Architect of the NSClient++ agent. He is an avid Open Source and monitoring fan and has been involved in open source monitoring for over 10 years. In Michael's day-to-day job, when he is not complaining about the lack of monitoring, he works as an Architect with Oracle Fusion Middleware. His blog, on which he often writes about monitoring, can be found at `http://blog.medin.name`.

Daniel Oetken, born near Hamburg, Germany in 1990, started using Linux in 2007. He spent 10 months in Vancouver, Canada and is now working as a Junior Server Administrator. He is working mostly with Debian web and database servers, and is responsible for the administration of a Splunk Enterprise Cluster.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?
- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

This book will show the readers how to setup an automated monitoring system for small to large-scale network and server infrastructure with Icinga that alerts them whenever something goes wrong in their network or the servers on the network. This book takes us through the installation, configuration, best practices, and deployment steps for Icinga, which is well configured to alert us with the most relevant and precise information whenever problems occur.

What this book covers

Chapter 1, *Installation and Configuration*, shows the dependency installation/configuration, followed by installation of Icinga (and its components) and the minimum configuration changes required to get the default Icinga installation up and running.

Chapter 2, *Icinga Object Configuration*, explains the default configuration files installed by the Icinga installation, and how the configuration objects bring together the localhost monitoring setup.

Chapter 3, *Running Remote Checks on Systems*, explains how to extend the existing understanding of the Icinga localhost monitoring configuration to monitor remote servers and network, including active/passive checks and NRPE/NSClient/SNMP.

Chapter 4, *Monitoring Infrastructure, Network Services, and System Health*, introduces the most common system health checks for Linux/Windows serves and network devices; and covers host-parent relationships and service dependencies.

Chapter 5, *Host and Service Availability Reporting*, covers notification configuration for host and service availability; explains the configuration directives related to specify to whom, when, and how Icinga should send notifications.

Chapter 6, *Icinga Plugins*, explains how to write the plugins (conventions with input, format of the output, exit code and so on) and integrate/use them with Icinga.

Chapter 7, *Web Interfaces*, introduces various web interfaces available for Icinga, and covers working with them and getting the most information out of them.

What you need for this book

To run the examples in the book, the following software will be required:

- Linux server:
    - ○ CentOS 6.x (`http://www.centos.org/`)
- Icinga (installation covered in the book):
    - ○ Icinga (latest version, `http://www.icinga.org/`)
- Miscellaneous tools:
    - ○ Email (SMTP) relay server (postfix/sendmail, postfix preferred, `http://postfix.org/`)
    - ○ Apache web (HTTP) server (`http://apache.org/`)

Who this book is for

This book is for all the system administrators or Linux enthusiasts who are looking for a flexible tool to monitor some kind of network infrastructure efficiently, or trying to understand the Icinga software. Readers are expected to have familiarity with Linux basics such as (package management, running a web server, and so on), and the Linux command line.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
define <object-type> {
  key1    value1
  key2    value2
  ...
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
define contactgroup {
    contactgroup_name       admins
    alias                   Icinga Administrators
    members                 icingaadmin
}
```

Any command-line input or output is written as follows:

```
$ cp icinga.spec ~/rpmbuild/SPECS
$ cp icinga-*.tar.gz ~/rpmbuild/SOURCES
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen".

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Installation and Configuration

Icinga is a scalable open source monitoring system that keeps a close watch on our network and server infrastructure, alerts us of problems and resolutions, and gives uptime reports. Icinga is a fork of the popular Nagios project, which aims to build many necessary features on top of it. Icinga is backward compatible with Nagios, so all of the Nagios configuration and plugins can be reused with Icinga as is.

This chapter covers a brief overview of the Icinga architecture and a basic setup to quickly start monitoring a localhost. We will look into the Icinga installation and gain some insight into the initial configuration, so as to get a basic deployment up and running. At the end of this chapter, we will have a web interface showing the status of various services on a localhost and also through an e-mail alert if one of the services goes down.

An overview

Icinga consists of various components. The Icinga Core handles scheduling of checks and processing their results. Service states are determined by the Core, by either running checks periodically (active checks) or checking the results that are being reported by a remote system (passive checks). Depending on the checks results, the service state is determined, and depending on the notification configuration, notifications are sent if there is a change in the service state.

The Core is not aware of the checks being executed, nor the notification method that is being used. It simply forks the processes to execute the check plugins, and also performs the processing of exit codes. The notification methods are defined by commands that invoke the underlying operating system provided commands (such as `sendmail`), or scripts that use notification services to actually deliver SMS, IMs, and so on; this means that Icinga is not aware of the particular notification methods either, it is only aware of the command/script to invoke or sending the notification.

The Core also provides all sorts of information (downtimes, check results, alert history, command execution logs, and so on) via logfiles, which can be used to generate reports on a web interface, for instance. Custom scripts and add-ons can use this logfile to read the current state, and data can show this information in various ways, which is useful for analysis, writing add-ons, and so on.

The data revealed by Icinga can be used to build various web interfaces to present monitoring state, and to do various actions from the web interface (schedule downtime, add comments, disable checks, and so on). There are already a few web interfaces available (Icinga Classic, Thruk, and so on) that can be plugged into Icinga and used.

There are many add-ons available online that extends Icinga's functionality to suit specific needs. Some available add-ons include NagVis (visualizing monitoring status), PNP4Nagios (graphing), and NConf and NagiosQL (managing configuration from web interface) . You can find a number of add-ons by visiting `http://www.monitoringexchange.org/`.

Icinga's configuration is stored in regular text files along with configuration objects in a simple-to-understand format. Each target server, checks to run on them, and so on are configuration objects that are related to each other as desired.

What to expect?

It is good to keep the outcome of the installation process in mind before beginning to have an overall idea of where we are headed, and what things will look like when we are done. We should expect the following at the end this chapter:

- Icinga installed with default configuration to monitor itself
- Icinga has started to monitor the localhost with most common checks (load, disk, number of logged-in users, HTTP, ping, SSH, swap, number of processes, and so on)
- Icinga web UI showing us the status of above checks
- E-mail notifications when a check fails

Requirements

If you are building Icinga from the source, you should have the GCC compiler and some libraries; Icinga will prompt for these during compilation if they are not found. Apart from that, you would need to have a functioning web server for the web interface.

We will need the following before we can start with the installation:

- GCC compiler
- Apache web server
- Nagios plugins (`nagios-plugins` package for most distributions)
- SMTP relay server (for example, Postfix) with proper configuration so that `mail/mailx` commands work

After we have these, we can move on to the download and installation steps.

Download

There are various sources and types of Icinga installers. You can get distribution-specific packages, like DEBs for Debian/Ubuntu, RPMs for Red Hat/CentOS, and so on. You can also get the source code and build packages yourself, or directly compile and install it. The source tarball can be downloaded from Icinga's official website from its **Downloads** section.

Installation

The recommended way of installing Icinga is by distribution packages. Debian (Squeeze/Wheezy) and Ubuntu already have upstream packages available on Launchpad, but the latest stable version may be old (Icinga is still relatively under development, so the latest version is preferred since it will have a lot of bug fixes). Red Hat/CentOS have Icinga packages located in the RepoForge YUM repository (`http://repoforge.org/`). It may still be useful to build an RPM for yourself, for using backported bug fixes without having to wait for the next release. Icinga's default source tarball already has the SPEC file for **RPM** creation.

Building an RPM for Red Hat / Centos

This is optional and should be done only if you want to build your own RPM. Otherwise, use the RepoForge repository mentioned previously and skip this section.

Icinga's source tarball has the SPEC file for building RPMs. The procedure to build RPMs remains as usual. Create a non-root user account, and log in with that user to build the RPMs. Do not do this as the root user.

Download the Icinga source tarball and extract it. Install RPM Development Tools using the following command:

```
$ sudo yum install rpmdevtools
```

You can also install it using the following command:

```
$ sudo yum install @development-tools fedora-packager
```

Now, set up the build folder:

```
$ rpmdev-setuptree
$ ls ~/rpmbuild
BUILD  RPMS  SOURCES  SPECS  SRPMS
```

Now, copy the SPEC file from the extracted source to SPECS and the source tarball to SOURCES:

```
$ cp icinga.spec ~/rpmbuild/SPECS
$ cp icinga-*.tar.gz ~/rpmbuild/SOURCES
```

Build the final RPM packages using the following commands:

```
$ cd ~/rpmbuild
$ rpmbuild -bb SPECS/icinga.spec
```

The preceding command might give some dependency errors. Install the listed packages with yum, and run the last rpmbuild command again. If all the dependencies are fixed, the compilation should start, which usually takes some time to complete. The RPM package will be created inside the RPMS/$arch folder, where $arch is the server's architecture (i386/x86_64).

```
$ ls ~/rpmbuild/RPMS/x86_64
icinga-1.9.1-1.el6.x86_64.rpm
icinga-idoutils-1.9.1-1.el6.x86_64.rpm
icinga-devel-1.9.1-1.el6.x86_64.rpm
icinga-idoutils-libdbi-mysql-1.9.1-1.el6.x86_64.rpm
icinga-doc-1.9.1-1.el6.x86_64.rpm
icinga-idoutils-libdbi-pgsql-1.9.1-1.el6.x86_64.rpm
icinga-gui-1.9.1-1.el6.x86_64.rpm
```

The icinga package is the Icinga Core; icinga-doc provides the Icinga offline documentation; icinga-gui provides the Icinga web interface; and icinga-idoutils-* provides an optional component to store Icinga information in a database. Now, install the icinga, icinga-doc, and icinga-gui built RPMs:

```
$ cd ~/rpmbuild/RPMS/x86_64/
$ sudo rpm –ivh {icinga,icinga-doc,icinga-gui}-*.rpm
```

The installation should succeed. Now install the Apache and Nagios plugins. The `nagios-plugins` package provides checking of all the basic check plugins that are commonly used for monitoring:

```
$ sudo yum install httpd nagios-plugins
```

Also, install SMTP relay server (Postfix is used in the next example) and use the `mail` command for sending e-mail alerts:

```
$ sudo yum install postfix mailx
```

Using DEB for Ubuntu

Debian packages for Ubuntu are available at `https://launchpad.net/ ubuntu/+source/icinga`, which can be used to install Icinga on an Ubuntu Server.

```
$ sudo apt-get install icinga icinga-doc
```

This command should install the Icinga package which is ready for use. Install the Apache and Nagios plugins:

```
$ sudo apt-get install apache2 nagios-plugins
```

Also use the `postfix` and `mail` commands:

```
$ sudo apt-get install postfix bsd-mailx
```

This setup is good enough for now. Let's continue with the next section.

Compiling from the source

Let us see how to compile Icinga from the source using the following steps:

1. Download the Icinga source tarball and extract it:

   ```
   $ tar zxvf icinga-1.9.1.tar.gz
   $ cd icinga-1.9.1
   ```

2. Run the configure script:

   ```
   $ ./configure --prefix=/usr --disable-idoutils
   ```

   IDOUtils is an optional module that comes with Icinga but we don't want to use it, so we use the `configure` script to disable the module compilation.

3. Compile the source code:

   ```
   $ make all
   ```

4. This will take some time. After it is done, it's time to install!

```
$ sudo make fullinstall
$ sudo make install-config
```

Icinga is now installed and (almost) ready for use.

We need to create a user named `icinga` because the Icinga server will run as this user:

```
$ useradd –m icinga
$ passwd icinga
```

Since the web server will be run as the `apache` user (in most cases, this is by default, and is not changed unless required by other applications using Apache), we need to add this user to the `icinga` group.

```
$ usermod –a icinga apache
```

The default web server user may be different for different distributions (for example, `www`, and so on).

Make it work

As mentioned in the *Requirements* section, make sure you have a proper SMTP relay server (such as Postfix) setup so that the usual `mail/mailx` commands can work. Verify this by sending yourself a test e-mail with the following command on the server (make sure you replace `your@email.com` with your own e-mail address):

```
$ echo "This is a test email" | /bin/mail –s "Test email"
  your@email.com
```

Proceed further if it works and you receive an e-mail. Troubleshoot your SMTP server if the preceding command gives an error, or you don't receive an e-mail. Open `/etc/icinga/objects/contacts.cfg`, and replace `icinga@localhost` with your own e-mail address in following line:

```
email                          your@email.com
```

By default, notifications are disabled, so we need to enable them first. Open `/etc/icinga/objects/localhost.cfg`, and comment all occurrences of the following line:

```
notifications_enabled          0
```

The configuration files can be commented with a hash (#) or a semicolon (;):

```
# notification_enabled        0
```

After the installation step, we now have Icinga Core, the web UI, and the Nagios plugins in place. The configuration to monitor your localhost for common services such as ping check, system load, and disk space is already in place. Now, start the `icinga` service, and (re)start the apache web server to begin the monitoring and see what's happening in the web interface. This can be done using the following commands.

- For RedHat/CentOS/Fedora:

  ```
  $ service icinga start
  $ service httpd start
  ```

- For Ubuntu/Debian:

  ```
  $ /etc/init.d/icinga start
  $ /etc/init.d/apache2 start
  ```

Make sure that there were no errors reported for the preceding commands. You can now access the web interface at `http://localhost/icinga`. The default authentication credentials are username: `icingaadmin` and password: `icingaadmin`.

> If you get a connection refused error, make sure that Apache was started properly. Also check if you have proper firewall settings in place to allow connections to the web server. This should not be a problem if you are already using the Apache server for other purposes.
>
> The web server may give constant "Internal Server Errors", due to some distributions shipping with SELinux which is enabled by default. We need to disable it to have the web functioning properly.
>
> ```
> $ setenforce 0
> ```

You can change the default password for the `icingaadmin` user to something of your choice with following command:

```
$ htpasswd /etc/icinga/passwd icingaadmin
```

This command will prompt for a new password to be set, type in the password and press *Enter*, this will save your password. Simply reload the web page, and it will ask for the new password.

You should have an interface similar to the following screenshot:

Icinga Web

Go to the **Service Detail** link on the left sidebar to see all the hosts (localhost only for now) and the services being monitored. It may take a few minutes for Icinga to schedule and complete the checking of all the services; after which, all of them will go green (given that they all pass, of course).

The next critical part is alerting. Note that we have a HTTP check for localhost (visible on the web interface). You can try to stop the web server and see if you get an e-mail notification. Note that it may take five to ten minutes before you get an e-mail alert. Icinga makes sure that the service is really down, by checking multiple times, before it sends out alerts. Also note that when we stop the web server, the web interface will remain inaccessible for that duration.

At this point, we're done with setting up a very basic monitoring server. The last section before we summarize the chapter provides some insight into the basic configuration options of Icinga, which typically applies to how Icinga operates rather than to what it monitors.

An overview of configuration options

Configuration is central to customizing Icinga to fit your needs. It becomes possible to highly customize Icinga since the configuration is completely text-based. There is no **out of the box** configuration that you can install, and automatically start discovering servers on your infrastructure and monitoring them. You need to carefully add all such configurations for Icinga to monitor your network efficiently. This topic gives you a brief overview on the Icinga configuration.

The Icinga configuration is split into two main types: resource files and object files, both of which are specified in the main configuration file along with other Icinga-wide options. This main file contains directives that determine how Icinga operates. The resource file contains Icinga macros that are user defined, such as paths to custom plugins, passwords, and so on. The object files contain Icinga object definitions, each object corresponding to a host, service, and so on. Icinga has object types (host, service, command, and so on) with parameters to build the monitoring configuration.

All of the configuration files are located inside the `/etc/icinga` folder. The commonly used main configuration file is named as `icinga.cfg`, the resource file is `resources.cfg` (it can be used more than once and is specified in the main configuration file). Object files are under the `/etc/icinga/objects` folder, which contains object definitions that specify monitoring targets. The names of the object files can be anything, with the content being object definitions, which we will look into later. Following is what the configuration structure looks like:

```
$ tree /etc/icinga
/etc/icinga
├── cgiauth.cfg
├── cgi.cfg
├── conf.d
├── icinga.cfg
├── modules
├── objects
│   ├── commands.cfg
│   ├── contacts.cfg
│   ├── localhost.cfg
│   ├── notifications.cfg
│   ├── printer.cfg
│   ├── switch.cfg
│   ├── templates.cfg
```

```
| ├── timeperiods.cfg
| └── windows.cfg
├── passwd
└── resource.cfg
```

We will look at what each configuration file refers to in the subsequent chapters. You can look up the Icinga documentation at `http://docs.icinga.org/latest/en/configmain.html` for the main configuration file, to see the list of options that are available to customize.

Summary

In this chapter, we looked into the basic overview of Icinga's monitoring architecture, installation, and running Icinga's server and web interface. We also looked into the overview of various aspects of configurations that exist in Icinga. Play around with the web interface and the configuration files to get comfortable with Icinga.

In the next chapter, we will go through and understand how the current configuration to monitor the localhost that is in place works.

2

Icinga Object Configuration

Objects are central to the Icinga configuration in terms of what will be monitored. We tell Icinga what servers and what services on each of these servers should be monitored, as well as the contact information and modes of alerting. All of these are primarily defined by Icinga objects.

It is very important to understand the objects and how to use them to build a proper monitoring configuration. This configuration is what we need to describe and configure our monitoring environment.

Objects

There are many types of objects, some of which include the following:

- Host, Hostgroup, Hostdependency, Hostescalation, Hostextinfo
- Service, Servicegroup, Servicedependency, Serviceescalation, Serviceextinfo
- Contact, Contactgroup
- Command
- Timeperiod

An object definition contains the object type and valid key-value pairs required to describe the particular object instance. The format of the object definition is as shown in the following code:

```
define <object-type> {
  key1    value1
  key2    value2
  ...
}
```

An example of object definition is as follows:

```
define host {
  hostname    www.google.com
  alias       google-server
}
```

Objects are defined in plain text files in a certain format. One file can contain one or more object definitions, and there can be any number of object configuration files with the extension `.cfg`, which are specified by the `cfg_file` and/or `cfg_dir` directives in the main configuration file.

The configuration of the basic monitoring setup involves the use of host, service, and command objects. We follow the step-by-step process to reach a final configuration which is as follows:

- We define servers that we want to monitor with host objects. The host objects take hostname, address, and so on, using which we specify the server's details.

- Then we define the services that will be monitored on each of the servers using service objects. The service objects take service name, hostname, check command, and so on to specify the server on which you want to run this service check, and the command name that specifies the command that will perform the check.

- The check command specified in the service definition corresponds to a command object. The command objects have a command name with the command-line string, which is specified to determine the exact command that should be run (as on the bash console) to execute the check. The criticality/state of the service check is determined by the exit code of the command: `0` is for **OK**, `1` is for **WARNING**, and `2` is for **CRITICAL**. All other exit codes result in the **UNKNOWN** state.

There are other object types available for easy configuration of complex networks. For example, suppose we want to run a check for free disk space on all of the servers, and we don't want to specify all of them in the service definition. Instead, we can create a hostgroup (basically a group of several hosts) object with a name, say `allservers`. We can then specify this hostgroup name in all host definitions using the `hostgroup` directive, and give the same hostgroup name in the service definition using the `hostgroup_name` directive. Icinga will put the service check on all hosts that belong to the `allservers` hostgroup. Even better, we can create a host object template (same as a usual host definition) using Icinga's object inheritance, in which we specify the desired hostgroup name and then simply inherit the template object in all host definitions using the `use` directive.

A localhost monitoring setup

Let us take a close look at our current setup, which we created in the first chapter, for monitoring a localhost. Icinga by default comes with object configuration for a localhost. The object configuration files are inside `/etc/icinga/objects` for default installations.

```
$ ls /etc/icinga/objects
commands.cfg    notifications.cfg    templates.cfg
contacts.cfg    printer.cfg          timeperiods.cfg
localhost.cfg   switch.cfg           windows.cfg
```

There are several configuration files with object definitions. Together, these object definitions define the monitoring setup for monitoring some services on a localhost.

Let's first look at `localhost.cfg`, which has most of the relevant configuration. We have a host definition:

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

The preceding object block defines one object, that is, the host that we want to monitor, with details such as the hostname, alias for the host, and the address of the server—which is optional, but is useful when you don't have DNS record for the hostname. We have a `localhost` host object defined in Icinga with the preceding object configuration.

The `localhost.cfg` file also has a hostgroup defined which is as follows:

```
define hostgroup {
    hostgroup_name  linux-servers
    alias           Linux Servers
    members         localhost     // host_name of the host object
}
```

The preceding object defines a hostgroup with only one member, `localhost`, which we will extend later to include more hosts. The `members` directive specifies the host members of the hostgroup. The value of this directive refers to the value of the `host_name` directive in the host definitions. It can be a comma-separated list of several hostnames. There is also a directive called `hostgroups` in the host object, where you can give a comma-separated list of names of the hostgroups that we want the host to be part of. For example, in this case, we could have omitted the `members` directive in the hostgroup definition and specified a `hostgroups` directive, which has the value `linux-servers`, in the `localhost` host definition.

At this point, we have a `localhost` host and a `linux-servers` hostgroup, and `localhost` is a member of `linux-servers`. This is illustrated in the following figure:
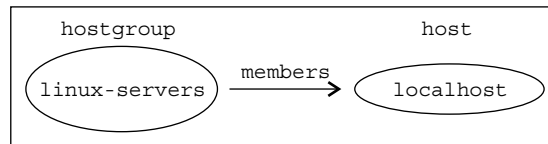
Going further into `localhost.cfg`, we have a bunch of service object definitions that follow. Each of these definitions indicate the service on a localhost that we want to monitor with the `host_name` directive.

```
define service {
    use                 local-service
    host_name           localhost
    service_description PING
    check_command       check_ping!100.0,20%!500.0,60%
}
```

This is one of the service definitions. The object defines a `PING` service check that monitors the reachability. The `host_name` directive specifies the host that this service check should be associated with, which in this case is `localhost`. Again, the value of the `host_name` directive here should reflect the value of the `host_name` directive defined in the host object definition. So, we have a `PING` service check defined for a localhost, which is illustrated by following figure:
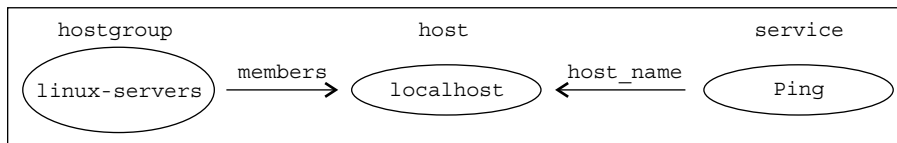
There are several such service definitions that are placed on a localhost. Each service has a `check_command` directive that specifies the command for monitoring that service.

> Note that the exclamation marks in the `check_command` values are the command argument separators. So, `cmd!foo!bar` indicates that the command is `cmd` with `foo` as its first argument and `bar` as the second.

It is important to remember that the `check_ping` part in `check_command` in the preceding example does not mean the `check_ping` executable that is in `/usr/lib64/nagios/plugins/check_ping` for most installations; it refers to the Icinga object of type command. In our setup, all command object definitions are inside `commands.cfg`.

The `commands.cfg` file has the command object definition for `check_ping`.

```
define command {
  command_name    check_ping
  command_line    $USER1$/check_ping -H $HOSTADDRESS$ -w $ARG1$ -c
  $ARG2$ -p 5
}
```

The `check_command` value in the `PING` service definition refers to the preceding command object, which indicates the exact command to be executed for performing the service check. `$USER1$` is a user-defined Icinga macro. Macros in Icinga are like variables that can be used in various object definitions to wrap data inside these variables. Some macros are predefined, while some are user defined. These user macros are usually defined in `/etc/icinga/resources.cfg`:

$USER1$=/usr/lib64/nagios/plugins

So replace the `$USER1$` macro with its value, and execute:

$ value/of/USER1/check_ping --help

This command will print the usual usage string with all the command-line options available. `$ARG1$` and `$ARG2$` in the command definition are macros referring to the arguments passed in the `check_command` value in the service definition, which are `100.0,20%` and `500.0,60%` respectively for the `PING` service definition. We will come to this later. As noted earlier, the status of the service is determined by the exit code of the command that is specified in the `command_line` directive in command definition.

We have many such service definitions for a localhost in `localhost.cfg`, such as `Root Partition` (monitors disk space), `Total Processes`, `Current Load`, `HTTP`, along with command definitions in `commands.cfg` for `check_commands` of each of these service definitions.

So, we have a host definition for `localhost`, a hostgroup definition `linux-servers` having `localhost` as its member, several service check definitions for `localhost` with check commands, and the command definitions specifying the exact command with arguments to execute for the checks. This is illustrated with the example `Ping` check in the following figure:
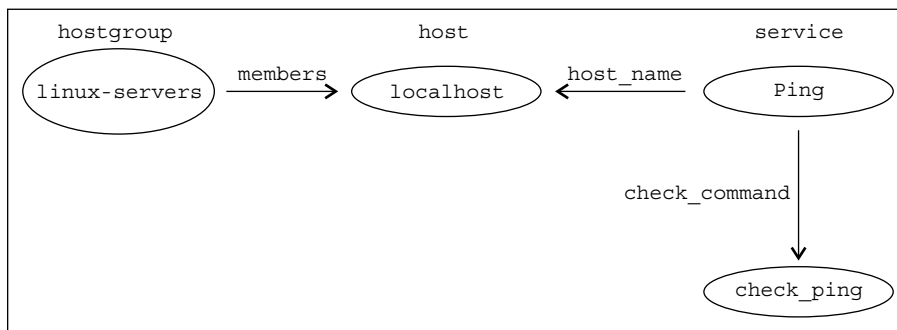
This completes the basic understanding of how our localhost monitoring is built up from plain-text configuration.

Templates

As you might have noticed, the host and the service object definitions have a `use` directive. Icinga's objects have the ability to inherit other objects. This `use` directive is nothing but the inheritance specifier. The value of this directive is another object of the same type, whose key-value pairs also are applied to the said object, overriding those which are re-defined. Such inherited objects are called **template objects**, or simply **templates**.

In our setup, the `localhost` host definition inherits the `linux-server` template object using the `use` directive. Template objects are usually defined in `templates.cfg`.

Remember that there is no specification of filenames inside /etc/ icinga/objects or the number of objects to be defined in each file. The only specification is that the extensions of the files should be .cfg.

```
define host{
  name                  linux-server
  use                   generic-host
  check_period          24x7
  check_interval        5
  retry_interval        1
  max_check_attempts    10
  check_command         check-host-alive
  notification_period   workhours
  notification_interval 120
  notification_options  d,u,r
  contact_groups        admins
  register              0
}
```

The preceding code is just another host object definition of linux-server, which is inherited by the localhost host definition. We indicate that this is not a real host by setting the value of the register directive to 0. This template object defines the most common directives that most of the host definitions are going to need. This reduces a lot of effort as we are no longer required to redefine all of these directives in all of the host definitions, we can just use the template. Note that this template in turn uses a template generic-host which has more general directives defined. It is defined in the same file.

Each host definition also has a check_command directive specified, which is used to monitor the state of the host itself, apart from all the service checks configured for it. It is the same as the check_command directive in service definitions. It refers to a command object which has the defined command_line. The commands.cfg file also has the command definition for check-host-alive, which is used here in the previous template.

Similarly, the service object definitions have a template service local-service that they're inheriting. This template service too is defined in templates.cfg.

Notifications

We would, as is the point of having monitoring systems, like to get alerted when something actually goes down. We don't want to keep monitoring the Icinga web interface screen, waiting for something to go down. Icinga provides a very generic and flexible way of sending out alerts. We can have any **alerting** script triggered when something goes wrong, which in turn may run commands for sending e-mails, SMS, Jabber messages, Twitter tweets, or practically anything that can be done from within a script. The default localhost monitoring setup has an e-mail alerting configuration, which we used in the first chapter.

The way these notifications work is that we define contact objects where we give the contact name, e-mail addresses, pager numbers, and other necessary details. These contact names are specified in the host/service templates or the objects themselves. So, when Icinga detects that a host/service has gone down, it will use this contact object to send contact details to the alerting script. The `contact` object definition also has the `host_notification_commands` and `service_notification_commands` directives. These directives specify the command objects that should be used to send out the notifications for that particular contact. The former directive is used when the host goes down, and the latter is used when a service goes down. The respective command objects are then looked up and the value of their `command_line` directive is executed. This command object is the same as the one we looked at previously for executing checks. The same command object type is used to also define notification commands.

> We can also define contact groups and specify them in the host/service object definitions to alert a bunch of contacts at the same time. We can also give a comma-separated list of contact names instead of a contact group.

Let's have a look at our current setup for notification configuration. The host/service template objects have the `admin` contact group specified, whose definition is in `contacts.cfg`:

```
define contactgroup {
  contactgroup_name    admins
  alias                Icinga Administrators
  members              icingaadmin
}
```

The group has the `icingaadmin` member contact, which is again defined in the same file:

```
define contact {
  contact_name    icingaadmin
  use             generic-contact
  alias           Icinga Admin
  email           your@email.com
}
```

The `contacts.cfg` file has your e-mail address. The `contact` object inherits the `generic-contact` template contact object.

```
define contact{
  name                          generic-contact
  service_notification_period   24x7
  host_notification_period      24x7
  service_notification_options  w,u,c,r,f,s
  host_notification_options     d,u,r,f,s
  service_notification_commands notify-service-by-email
  host_notification_commands    notify-host-by-email
  register                      0
}
```

This template object has the `host_notification_commands` and `service_notification_commands` directives defined as `notify-host-by-email` and `notify-service-by-email` respectively. These are commands similar to what we use in service definitions. These commands are defined in `commands.cfg`:

```
define command {
  command_name    notify-host-by-email
  command_line    /usr/bin/printf "%b" "***** Icinga
  *****\n\nNotification Type: $NOTIFICATIONTYPE$\nHost:
  $HOSTNAME$\nState: $HOSTSTATE$\nAddress: $HOSTADDRESS$\nInfo:
  $HOSTOUTPUT$\n\nDate/Time: $LONGDATETIME$\n" | /bin/mail -s "**
  $NOTIFICATIONTYPE$ Host Alert: $HOSTNAME$ is $HOSTSTATE$ **"
  $CONTACTEMAIL$
}
define command {
  command_name    notify-service-by-email
  command_line    /usr/bin/printf "%b" "***** Icinga
  *****\n\nNotification Type: $NOTIFICATIONTYPE$\n\nService:
  $SERVICEDESC$\nHost: $HOSTALIAS$\nAddress: $HOSTADDRESS$\nState:
  $SERVICESTATE$\n\nDate/Time: $LONGDATETIME$\n\nAdditional
  Info:\n\n$SERVICEOUTPUT$\n" | /bin/mail -s "**
  $NOTIFICATIONTYPE$ Service Alert: $HOSTALIAS$/$SERVICEDESC$ is
  $SERVICESTATE$ **" $CONTACTEMAIL$
}
```

These commands are eventually executed to send out e-mail notifications to the supplied e-mail addresses. Notice that `command_lines` uses the `/bin/mail` command to send e-mails, which is why we need a working setup of a SMTP server. Similarly, we could use any command/script path to send out custom alerts, such as SMS and Jabber. We could also change the above e-mail command to change the content format to suit our requirements.

The following figure illustrates the contact and notification configuration:
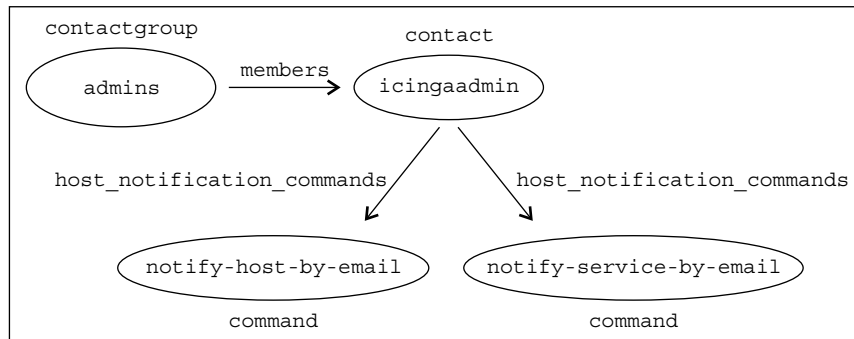
The correlation between hosts/services and contacts/notification commands is shown below:
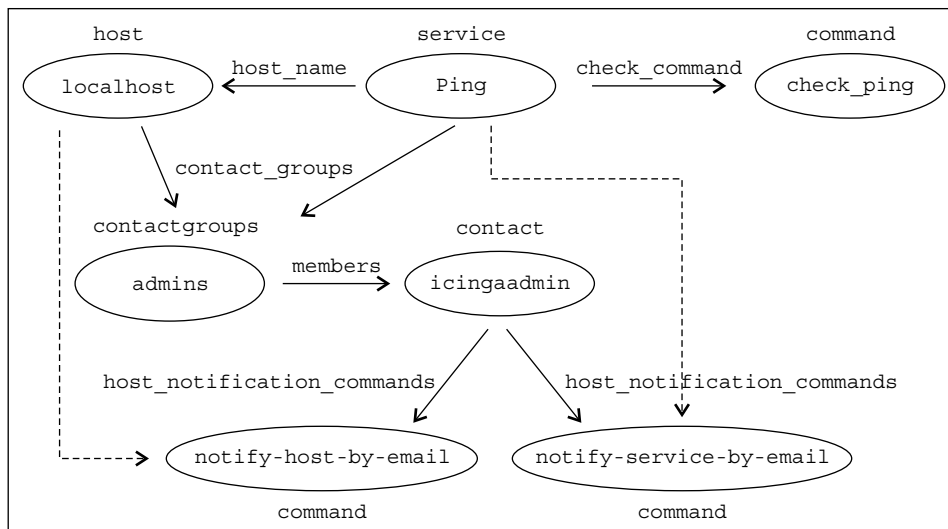
Summary

In this chapter, we analyzed our current configuration for the Icinga setup which monitors a localhost. We can replicate this to monitor a number of other servers using the desired service checks. We also looked at how the alerting configuration works to send out notifications when something goes down.

In the next chapter, we take a look at how we can extend the current configuration setup to monitor remote servers and run checks remotely or from remote servers.

3

Running Remote Checks on Systems

So far, we have seen how to define service checks for a localhost. But the real use case of a monitoring server like Icinga is to monitor an entire infrastructure, not to deploy Icinga on each of the hosts you would want to monitor. This chapter covers ways to monitor remote servers from an Icinga instance, similar to the one we have for localhost monitoring. A similar configuration is used to monitor remote servers as well, with slight modifications.

There are several different ways of monitoring our infrastructure using remote servers, depending on the needs and services we want to monitor.

- **Active checks**: The monitoring server polls the remote server at fixed intervals of time to check the status of the service. For example, the Icinga server would periodically run a command to make a test HTTP connection to the remote host to fetch the status of the HTTP check.
- **Passive checks**: The remote hosts check the status of the service themselves and submit it to the monitoring server. The hosts would have to report both critical and recovery events to the monitoring server.

The type of the check can be configured on a per-service check basis using the `active_checks_enabled` and `passive_checks_enabled` directives in the service object definition.

Both active and passive checks have their appropriate use cases. It is important to determine the type of check to be used for your use case. Active checking is generally recommended for monitoring services such as HTTP, IMAP, and so on; while passive checking is recommended for services that are long running or are generated by internal events of the host, such as monitoring a logfile for errors; such an event would submit a CRITICAL event to Icinga.

Further, we will have a look at what tools are required for both types of checks.

Active checks

The monitoring server initiates the checks at specific intervals and their statuses are set according to the return value of the check plugin. There are several ways to retrieve status of a service, depending on the kind of service it is.

There are majorly two types of services: public services and private services. We will look at both of them in this section.

Public services

Publicly available services include services that are accessible over the network, either the internal network or via the Internet; basically, ones that can be checked by establishing the network connection and optionally making a sample request. Examples include HTTP, FTP, SSH, IMAP, SMTP, and MySQL Server.

If, for example, we want to monitor HTTP, SSH, and IMAP services on server1.example.org, which is some remote host other than the monitoring server itself, the host and services configuration would look like the following:

- Host definition:

```
define host {
  use         linux-server
  host_name   server1.example.org
  alias       Example server 1
  address     172.16.143.22
  hostgroups  linux              ; just an example
}
```

Icinga's default set of configuration comes with a linux-server host template, which is defined in templates.cfg. Following is the configuration for a few service checks.

- HTTP, which makes a GET / request on port 80:

```
define command {
  command_name    check_http
  command_line    $USER1$/check_http -I $HOSTADDRESS$
}

define service {
  use                   generic-service
  host_name             server1.example.org
  service_description   HTTP
  check_command         check_http
}
```

- SSH:

```
define command {
  command_name    check_ssh
  command_line    $USER1$/check_ssh $ARG1$ $HOSTADDRESS$
}

define service {
  use                 generic-service
  host_name           server1.example.org
  service_description SSH
  check_command       check_ssh
}
```

- IMAP:

```
define command {
  command_name    check_imap
  command_line    $USER1$/check_imap -H $HOSTADDRESS$ $ARG1$
}

define service {
  use                 generic-service
  host_name           server1.example.org
  service_description IMAP
  check_command       check_imap
}
```

We can play around with command-line arguments that the check plugin such as `check_http` provides; for example, warning/critical threshold values of response time, and so on. We will cover the check plugins in detail in next chapter.

> Icinga service must be reloaded whenever we update any of the configuration files for these changes to take effect.
>
> **$ sudo service icinga reload**
>
> The `reload`/`restart` command verifies the entire configuration for syntax or semantics errors, and reports any errors found. It is recommended, as a general practice, to always do a configuration check before reloading/restarting Icinga.
>
> **$ sudo service icinga show-errors**
>
> The preceding command verifies the Icinga configuration and shows the errors, if any.

Private services

Private services include various system resource and performance checks, such as checks for free disk space, CPU load, memory usage, number of processes, and so on. Such information is not available over the network and has to be acquired using some intermediate agents that can provide the same when requested. Some of the agents are as follows:

- SSH (Linux servers)
- NRPE (Linux servers)
- NSClient++ (Windows servers)
- SNMP (routers, switches, and so on)

These agents can also be used to check for public services that are not necessarily accessible from the Icinga server, or the purpose of the check is different. For example, to test the reachability of web server running on server1 from server2 (both of which are different from Icinga server), running the HTTP check for server1 from Icinga server would not serve the purpose. This use case will use one or more of these agents running on server2 so that they can provide Icinga with the status of reachability of server1.

Secure Shell (SSH)

The simplest way to get any information from a Linux server is to run SSH on the remote server and run any command/script to get the information. The `nagios-plugins` package provides ready-to-use plugins for such purposes (`check_load`, `check_disk`, and so on). We have used these in our localhost monitoring setup. So, we can define a command to SSH the server, run one of these plugins, and return the output, which is determined to set the status of the service check on the monitoring server. We also need to ensure that the `nagios-plugins` package is installed on the remote server, so that we have all the available check plugins to execute over SSH. Let's look at a configuration for disk space check on `server1.example.org`:

```
define command {
  command_name     check_by_ssh
  command_line     $USER1$/check_by_ssh -H $HOSTADDRESS$ -C
  'PATH=$PATH:/usr/lib64/nagios/plugins $ARG1$'
}

define service {
  use                    generic-service
```

```
host_name               server1.example.org
service_description     Disk
check_command           check_by_ssh!check_disk -w 20% -c 10%
}
```

Icinga runs and executes checks as user configured using the `icinga_user` directive in `icinga.cfg`. So, we need to make sure that proper SSH keys are generated for the user on the monitoring server, and added to `authorized_keys` of the remote server(s) so that `check_by_ssh` can execute flawlessly.

To generate SSH keys for the `icinga` user, use the following command:

```
$ su icinga -c 'ssh-keygen -t rsa'
```

Keep pressing *Enter* to give default values for presented parameters. This will generate the SSH public key (`~/.ssh/id_rsa.pub`) and the SSH private key (`~/.ssh/id_rsa`) in the `.ssh` directory inside the home folder of the `icinga` user.

It is necessary to put the public key in `~/.ssh/authorized_keys`, which is in the home folder of the `icinga` user on the remote host. You will have to make sure the `icinga` user exists on the remote host. This will give SSH access to the `icinga` user on the Icinga server for the `icinga` user on the remote host.

We appended `/usr/lib64/nagios/plugins` to `PATH` so that the `check_by_ssh` command object can be re-used to run other plugins over SSH, without having to give the full path in the command every time.

Nagios Remote Plugin Executor (NRPE)

NRPE is an add-on that is deployed on the remote hosts to execute the check plugins on them. It is similar to using SSH; NRPE daemon has to be running on the remote server and its configuration should have a `command_name` to `command_executable` (with arguments) mapping. So, when Icinga executes the `check_nrpe` check, it uses the NRPE command name that we specify in the service definition, then sends it to the NRPE agent (daemon) on the remote server. This executes the corresponding command line and returns the exit code.

There are pros and cons in using this method instead of using the SSH method. SSH gives us more flexibility in terms of running any desired command or script over SSH. NRPE has an overhead of defining NRPE command-name to command-executable mapping and other required configuration. On the other hand, SSH increases the load on the monitoring server if there are a large number of checks, due to frequent opening and closing of SSH connections.

Each execution of a check calls for a SSH connection, execution, and closing of the connection, which is a considerable overhead. Following is an example of the NRPE daemon configuration (usually `/etc/nrpe.cfg`), similar command and service object definitions can be used to execute checks over NRPE:

```
# command[<command_name>]=<command_line>
command[check_users]=/usr/lib64/nagios/plugins/check_users -c 10
command[check_load]=/usr/lib64/nagios/plugins/check_load -c 40%
```

More information on installation and configuration of NRPE can be found at `http://docs.icinga.org/latest/en/nrpe.html`.

NSClient++

While the above methods are best suited for Linux servers, they are not supported for the Windows servers. For this purpose, there is an agent called NSClient++. It is the Windows' replacement for Linux's NRPE daemon, although it is cross-platform and available for Linux too. The same `check_nrpe` plugin can be used to run commands on remote Windows servers. The plugin contacts the NSClient++ agent and asks for the status of one of the commands made available by the agent. The list of available commands and their usage can be found in the agent's documentation.

For example, if we have a Windows server with the hostname `server2.example.org`, the host definition can be as follows:

```
define host {
  use            windows-server
  host_name      server2.example.org
  alias          Example server 2
  address        172.16.143.23
  hostgroups     windows              ; just an example
}
```

Icinga already provides a `windows-server` host template, found in `templates.cfg`. Following is what the Icinga configuration for checking CPU load would look like (NSClient++ supports a command called `CheckCPU`):

```
define command {
  command_name    check_nrpe
  command_line    $USER1$/check_nrpe -u -H $HOSTADDRESS$ -c $ARG1$
  -a $ARG2$
}

define service {
  use                    generic-service
```

```
    host_name                  server2.example.org
    service_description    CPU
    check_command              check_nrpe!CheckCPU!warn=80 crit=90
}
```

We'd need a working NSClient++ deployed on the remote Windows server(s) for the preceding code to work. Have a look at `http://docs.icinga.org/latest/en/monitoring-windows.html#installwindowsagent` for the same. Make sure proper whitelisting is done on the Windows server side to allow `check_nrpe` to talk to the agent. The list of commands supported by NSClient++ is available at `http://www.nsclient.org/nscp/wiki/CheckCommands`.

Simple Network Management Protocol (SNMP)

SNMP agents on routers and switches can be used to monitor checkpoints or services on them. Monitoring network devices mostly includes simply network traffic and open ports check. A wide range of such values is available via SNMP and are called the OIDs. Each **Object Identifier** (**OID**) has a value associated with it. For example, the OID `sysUpTime.0` gives the uptime of the device. An example of a host definition for a router is as follows:

```
define host {
  use           generic-switch
  host_name     switch1.example.org
  alias         HP 12504 AC switch
  address       192.168.32.58
  hostgroups    switches              ; just an example
}
```

An example of a service check for getting the uptime is as follows:

```
define command {
  command_name    check_snmp
  command_line    $USER1$/check_snmp -H $HOSTADDRESS$ -o $ARG1$
  $ARG2$
}

define service {
  use                    generic-service
  host_name              switch1.example.org
  service_description    SNMP
  check_command          check_snmp!sysUpTime.0
}
```

Readers are advised to read on how to use SNMP to get various kinds of values. These values can then be used for monitoring. The `check_snmp` plugin is used to contact the device and query the values via the SNMP agent after supplying relevant authorization. The `snmpwalk` command can be used to get the list of available OIDs with a particular device:

```
$ snmpwalk -mAll –v1 –cpublic switch1.example.org system
```

The previous command gives you a list of all OIDs and their values as reported by the switch device.

Passive checks

Checks that are run by the remote hosts themselves and the status is submitted to the Icinga monitoring server are classified as passive checks. In simpler terms, Icinga does not actively execute the checks; it just sits and waits for hosts to submit the check status periodically.

While active checks are a sure way of getting up-to-date status of the services, passive checks are useful in many cases to offload a part of the monitoring to the remote servers themselves. Also, passive checks are often used to monitor only the private services; it does not make sense to check for the network (public) services on servers themselves. External reachability and status should be checked by the monitoring server only.

We need to ensure two things in configuration to enable passive checks:

- The `accept_passive_service_checks` directive in `icinga.cfg` is set to `1`
- The `passive_checks_enabled` directive is set to `1` in service definition

We would need to have service definitions for passive checks in the same format as we have for other usual checks, with just one added directive as noted previously. Although Icinga won't do anything about the passive checks, but it needs to know that it should expect the check results for that service—hence the service definition.

We also require agents to be deployed on both Icinga and remote servers. The following are the required agents:

- Icinga server requires a NSCA add-on, such as the Nagios Service Check Acceptor provided by the `nagios-nsca` package, which runs as a server listening for connections, accepts check results from remote hosts and writes them to Icinga's processing queue.

- The remote hosts need to have a NSCA client installed. In Linux, it is provided by the `nagios-nsca-client` package, and in Windows you can use the NSClient++. The NSCA client submits the given check result to the given NSCA server.

> The NSCA daemon and client versions should not vary too much. There have been compatibility issues with encryption/decryption.

The remote server executes the check script periodically, using cron jobs (for Linux) or Task Scheduler (for Windows) or when a relevant event is triggered. Then, it executes the `send_nsca` command, provided by the NSCA client package, that takes a string from standard input in the following format and sends it to the Icinga server to report the check results:

```
<host_name>[tab]<svc_description>[tab]<return_code>[tab]<plugin_
output>[newline]
```

In the preceding line, the description of the parameters is as follows:

- `host_name` is the hostname of the server, which is defined in the host definition, and on which the check is running
- `svc_description` is the service description of the service check, which is defined in the service definition
- `return_code` is one of the exit codes (`0` is OK, `1` is WARNING, and `2` is CRITICAL)
- `plugin_output` is the additional information used for debugging the problem

An example of this is given in the next section.

The NSCA client, which is called by the remote scripts, submits the check results by connecting to the NSCA server. Then the server forwards the check results to Icinga after doing some basic validation.

Let's look at an example configuration for adding a passive check to monitor whether `/home` exists on `server1.example.org`:

```
define service {
  use                     generic-service
  host_name               server1.example.org
  service_description     Data
  active_checks_enabled   0
  normal_check_interval   30
  check_command           check_dummy!2!Check result not
  received
}
```

Note that the template service `generic-service` has both `active_checks_enabled` and `passive_checks_enabled` set to `1`, so we need to set the former to `0` in order to make it strictly a passive check. The script on the remote server is to be run every 30 minutes, so we set `normal_check_interval` to `30`. Also, we use `check_dummy` in the `check` command to make the check fail if check results are not submitted within the last `normal_check_interval` minutes. This is done to make sure of the freshness of the present check status. See the `help` section of the `check_dummy` check plugin to know its usage.

The `check` script to check for existence of a directory, which will be run on the remote server, is as follows:

```
NSCA_HOST="icinga.example.org"  # Icinga server running the NSCA
  server
NSCA_CONF="/etc/nagios/send_nsca.cfg"

if [[ -d /var/data ]]; then
    ret=0
    output="/var/data exists"
else
    ret=2
    output="/var/data does not exist"
fi

echo "server1.example.org\tData\t$ret\t$output\n" |
  /usr/bin/send_nsca -H $NSCA_HOST -c $NSCA_CONF
```

This `check` script is run as a cron job on `server1.example.org` and will keep submitting check results every 30 minutes. Note that `server1.example.org` and `Data` in the echo string passed to `send_nsca` should match the `host_name` directive in the host definition and the `service_description` directive in service definition respectively.

Summary

In this chapter, we looked at various ways of running service checks on remote servers and the use cases where these ways can be useful. In general, passive checks are less commonly used compared to active checks. But the checks that are expected to run for a relatively longer time before they return the status should be setup as passive checks. As a rule of thumb, publicly available services should be monitored with active checks, while the rest can be over SSH/NRPE or passive checks depending on the nature of the service checks. As a fallback method, whenever it is required to run checks remotely, checking by SSH is the most flexible and preferred way of executing checks remotely.

We will take a look at the most common system resource and performance checks for Linux and Windows hosts, and network devices, so as to ensure proper health of our servers.

4

Monitoring Infrastructure, Network Services, and System Health

In this chapter, we will look into common system health check techniques (private services) for Windows and Linux servers and network devices, along with their configuration, and setting parent-child relationships between Icinga hosts and services.

The `nagios-plugins` package provides many check plugins for checking many common things. More plugins can be found online for specific use cases. Nagios Exchange (`http://exchange.nagios.org/directory/Plugins`) and MonitoringExchange (`http://www.monitoringexchange.org/`) are some of the very resourceful sources for useful plugins. Some plugins are even provided as distribution packages. It is required to install the `nagios-plugins` packages on both the Icinga server and the servers to be monitored — at the server side to run checks for public services from the server, and at the host's side to run checks over an agent on hosts. Most of the plugins are installed inside `/usr/lib/nagios/plugins` or `/usr/lib64/nagios/plugins`, depending on the machine's architecture.

In this chapter, we will take into consideration the following hostgroups:

- The `linux` hostgroup is used for all Linux servers.

```
define hostgroup {
    hostgroup_name      linux
    alias               Linux servers
}
```

- The `windows` hostgroup is used for all Windows servers.

```
define hostgroup {
    hostgroup_name      windows
    alias               Windows servers
}
```

- The `switches` hostgroup is used for all switches (the checks would also apply to routers).

```
define hostgroup {
    hostgroup_name      switches
    alias               Network switches
}
```

Any publicly available service (accessible over an internal network, or the Internet from the Icinga monitoring server) can be easily monitored irrespective of the operating system of the server hosting it, given that proper whitelisting is for the Icinga server in firewalls and other security software. We have covered the monitoring of the public services in the previous chapter, so this chapter will cover the monitoring of the common private services (such as CPU load, disk space, and number of processes) for different operating systems.

Linux servers

Let's add an example Linux server's host object to the `linux` hostgroup.

```
define host {
    use             linux-server
    host_name       server1.example.org
    address         192.168.32.56
    hostgroups      all,linux
}
```

Similarly, we add the `hostgroups` directive in all our Linux servers' host definitions.

We will use `check_by_ssh` to perform the private service checks. The common command definition is as follows:

```
define command {
    command_name    check_by_ssh
    command_line    $USER1$/check_by_ssh -H $HOSTADDRESS$ -C
'PATH=$PATH:/usr/lib64/nagios/plugins $ARG1$'
}
```

The common service checks for Linux servers include:

- The **Secure Shell** (**SSH**) check
- The load check
- The disk check

The Secure Shell (SSH) check

Although the SSH check is a publicly available service, it is important to mention this service check here because all the `check_by_ssh` checks rely on this SSH service, and it's a good idea to place a check for SSH itself.

Just a quick look on the SSH check(note that it is `check_ssh` not `check_by_ssh`):

```
define command {
    command_name    check_ssh
    command_line    $USER1$/check_ssh -H $HOSTADDRESS$
}
define service {
    use                  generic-service
    hostgroup_name       linux
    service_description  SSH
    check_command        check_ssh
}
```

This service check would generate alerts, which we will notice on the web interface, if there were a problem with getting SSH access into the server and subsequently. All checks relying on it will also start to fail, that is, generate an alert.

The load check

The `check_load` plugin is provided under the standard directory as mentioned earlier. It takes warning (as a `-w` switch) and critical load values (as a `-c` switch) and returns the corresponding exit status for the currently reported system load averages for the past 1, 5 and 15 minutes. The service definition is as follows:

```
define service {
    use                 generic-service
    hostgroup_name      linux
    service_description Load
    check_command       check_by_ssh!check_load -w 1,7,11 -c 2,10,15
}
```

Alternatively, we can define a wrapper `check_load_by_ssh` command object to be able to re-use it generically in other service definitions for some host which is not in the `linux` hostgroup.

```
define command {
    command_name            check_load_by_ssh
    command_line            $USER1$/check_by_ssh -H $HOSTADDRESS$ -C
'$USER1$/check_load -w $ARG1$ -c $ARG2$ $ARG3$'
}

define service {
    use                 generic-service
    hostgroup_name      linux
    service_description Load
    check_command       check_load_by_ssh!1,7,11!2,10,15
}
```

This service check will give:

- CRITICAL for load averages of more than 2,10,15
- WARNING for load averages of more than 1,7,11
- OK for load averages of less than 1,7,11

We can also divide the load averages by the number of CPU cores using the `-r` switch in `check_load`, and set warning or critical thresholds on these load averages.

The disk check

The `check_disk` plugin is available as part of the standard `nagios plugins` package. It allows us to set WARNING and CRITICAL thresholds for the free disk space, in terms of specific amount of disk space or a percentage.

```
define command {
    command_name              check_disk_by_ssh
    command_line              $USER1$/check_by_ssh -H $HOSTADDRESS$ -C
'$USER1$/check_disk -w $ARG1$ -c $ARG2$ $ARG3$'
}

define service {
    use                   generic-service
    hostgroup_name        linux
    service_description   Disk
    check_command         check_disk_by_ssh!20%!10%
}
```

This would generate:

- CRITICAL for less than 10 percent of the free disk space
- WARNING for less than 20 percent of the free disk space
- OK for more than 20 percent of the free disk space

The plugin also provides `-W/--iwarning` and `-C/--icritical` switches to check for percentage of free inode space. We can also specify a particular path (`--path`), partition (`--partition`), or a mount point (`--mountpoint`) to check for free disk/inode space.

Windows servers

Let's add an example Windows server's host object to the `windows` hostgroup.

```
define host {
    use            windows-server
    host_name      server2.example.org
    address        192.168.32.57
    hostgroups     all,windows
}
```

Similarly, we add the `hostgroups` directive in all our Windows servers' host definitions. We will use `check_nrpe` with NSClient++ agent to monitor private services on Windows servers. The common command definition is as follows:

```
define command {
    command_name    check_nrpe
    command_line    $USER1$/check_nrpe -u -H $HOSTADDRESS$ -c $ARG1$
-a $ARG2$
}
```

Common checks for Windows servers include:

- The **Nagios Remote Plugin Executor** (**NRPE**) check
- The CPU check
- The memory check
- The disk check

The Nagios Remote Plugin Executor (NRPE) check

Since we rely on NRPE to deliver data related to check results, it is highly recommended to add a service check for NRPE itself to make sure it is working fine. Note that this is also a public service (just as SSH is), and can be monitored without an agent.

```
define command {
    command_name            check_nrpe_status
    command_line            $USER1$/check_nrpe -H $HOSTADDRESS$ $ARG1$
}

define service {
    use                     generic-service
    hostgroup_name          windows
    service_description     NRPE
    check_command           check_nrpe_status
}
```

The service check will generate alert if NRPE is not working, and all other checks relying on it will also start to fail.

The CPU check

NSClient++ provides a `CheckCPU` command under the `CheckSystem` category.
It accepts warning and critical thresholds in percentage.

```
define service {
    use                    generic-service
    hostgroup_name         windows
    service_description    CPU
    check_command          check_nrpe!CheckCPU!warn=50 crit=60
time=20m time=10s time=4
}
```

Alternatively, we could write a wrapper command definition for the CPU check:

```
define command {
    command_line           Check CPU
    command_name           $USER1$/check_nrpe -u -H $HOSTADDRESS$ -c
CheckCPU -a warn=$ARG1$ crit=$ARG2$ $ARG3$
}

define service {
    use                    generic-service
    hostgroup_name         windows
    service_description    CPU
    check_command          CheckCPU!40!60!time=20m time=10s time=4
}
```

The preceding check will give:

- CRITICAL for more than has 60 percent CPU load
- WARNING for more than 40 percent CPU load
- OK for less than 40 percent CPU load

The multiple time parameters indicate different time periods over which load
averages have to be calculated. So, we have multiple CPU load summaries and
warning/critical thresholds applied to each of them.

The memory check

The `CheckMEM` command of NSClient++ is also under the `CheckSystem` modules available in the standard checks plugin.

```
define command {
    command_line            CheckMem
    command_name            $USER1$/check_nrpe -u -H $HOSTADDRESS$ -c
CheckMEM -a MaxWarn=$ARG1$ MaxCrit=$ARG2$ $ARG3$
}

define service {
    use                     generic-service
    hostgroup_name          windows
    service_description     Memory
    check_command           CheckMEM!50%!60%
}
```

This check gives:

- CRITICAL for more than 60 percent of used memory
- WARNING for more than 50 percent used memory
- OK for less than 50 percent used memory

The `CheckMEM` command also provides the `type` option to specify what type of memory we want to monitor. Valid values are page (available page memory), paged (used page memory), virtual (used number of pages of swap), or physical (available physical memory). The last two are used most commonly.

The disk check

The `CheckDisk` module is another module of the check commands that includes commands such as `CheckFileSize` and `CheckDriveSize`. Let us define a check for the size of the `Windows` folder.

```
define command {
  command_name            CheckFileSize
  command_line            $USER1$/check_nrpe -H $HOSTADDRESS$
-c CheckFileSize -a ShowAll MaxWarn=$ARG1$ MaxCrit=$ARG2$
File:$ARG3$=$ARG4$
}

define service {
  use                     generic-service
  hostgroup_name          windows
  service_description     Windows Folder Size
  check_command           CheckFileSize!1024M!4096M!_WIN!c:/WINDOWS/*.*
}
```

The preceding service check would generate alerts for the folder size of _WIN, defined as c:/WINDOWS/*.*:

- CRITICAL for size of more than 4096 MB (4 GB)
- WARNING for size of more than 1024 MB (1 GB)
- OK for size of less than 1024 MB (1 GB)

Similarly, to monitor the size of swap space (the pagefile.sys file):

```
define command {
    command_name            CheckPageFile
    command_line            $USE1$/check_nrpe -H $HOSTADDRESS$ -c
CheckFileSize -a ShowAll MinWarn=$ARG2$ MinCrit=$ARG1$ File=c:/
pagefile.sys
}

define service {
    use                     generic-service
    hostgroup_name          windows
    service_description     Pagefile.sys Filesize
    check_command           CheckPageFile!1024M!512M
}
```

The service check would generate alerts for file size of pagefile.sys:

- CRITICAL for less than 1024 MB (1 GB) of the file size
- WARNING for less than 512 MB of the file size
- OK for size of more than 512 MB of the file size

Note that we have used MinWarn and MinCrit to indicate that we want to get alerted if the size becomes *less than* the thresholds. Use MaxWarn and MaxCrit for alerts when size becomes *more than* the thresholds.

The check configuration for monitoring free disk space on drive(s):

```
define command {
    command_name            CheckDriveSize
    command_line            $USER1$/check_nrpe -H $HOSTADDRESS$ -c
CheckDriveSize -a ShowAll CheckAll MinWarn=$ARG1$ MinCrit=$ARG2$
}

define service {
    use                     generic-service
    hostgroup_name          windows
    service_description     Disk Volumes
    check_command           CheckDriveSize!20%!10%
}
```

So this service check would generate:

- CRITICAL for less than 10 percent of the free disk space
- WARNING for less than 20 percent of the free disk space
- OK for more than 20 percent of the free disk space

> For the `CheckDriveSize` command, the values of the `MaxWarn` and `MaxCrit` options are for the maximum *used disk space* thresholds; whereas `MinWarn` and `MinCrit` are for the minimum *free disk space* thresholds.

You can use the `Drive` attribute to specify the particular drive whose size you want to monitor.

Network devices

Let's first define a host for a `sw-1.example.org` switch. Icinga already provides a template host for switches.

```
define host {
    use             generic-switch
    host_name       hp-ac12504
    alias           HP 12504 AC switch
    address         192.168.32.58
    hostgroups      all,switches
}
```

Icinga gives a `generic-switch` template host found in `templates.cfg` with switch specific tweaks. Similarly, we add all switches to the `switches` hostgroup using the `hostgroups` directive in their host definitions.

Most routers and switches support **Simple Network Management Protocol** (**SNMP**) for monitoring. SNMP exposes data of the device in the form of variables, which can be queried remotely.

We will use the `check_snmp` check plugin provided by the default `nagios plugins` installation to execute check commands for monitoring the network devices. The common command definition is as follows:

```
define command {
    command_name        check_snmp
    command_line        $USER1$/check_snmp -H $HOSTADDRESS$ -o $ARG1$
$ARG2$
}
```

The identification scheme to identify variables that expose data is called OID in SNMP. The OID we want to query is passed using the `-o` command-line option to `check_snmp`. So we have made it the first argument to the `check_snmp` Icinga command definition and rest of the command-line arguments can be passed separately.

The packet loss and RTA check

The packet loss and RTA check is simply a packet loss (ping) and round trip average time (average time taken by packet to make a round trip from Icinga server to the network device) check between Icinga monitoring server and the switch device. The `check_ping` command already provides this checking functionality, so we won't need to use `check_snmp` for this particular check, instead we use `check_ping`.

```
define command {
    command_name        check_ping
    command_line        $USER1$/check_ping -H $HOSTADDRESS$ -w $ARG1$
-c $ARG2$ $ARG3$
}

define service {
    use                 generic-service
    hostgroup_name      switches
    service_description Ping
    check_command       check_ping!300,5%!700,20%
}
```

The service check will generate:

- CRITICAL for more than 700 ms RTA or more than 20 percent packet loss
- WARNING for more than 300 ms RTA or more than 5 percent packet loss
- OK for less than 300 ms RTA and less than 5 percent packet loss

Note that we can pass more command-line options to `check_ping` using a third argument in the `check_command` directive of service definition, for example, to specify the timeout for the check plugin.

The SNMP status

If we are to monitor switch devices using SNMP, it is important to add a check for SNMP itself. We will use the simple uptime command via SNMP to check its status. If SNMP fails, the uptime command will also fail resulting in an alert.

```
define service {
    use                 generic-service
    hostgroup_name      switches
    service_description SNMP
    check_command       check_snmp!sysUpTime.0
}
```

In the preceding code, `sysUpTime.0` is a SNMP OID for getting the uptime value. If this service check fails, all other service checks relying on SNMP will also start failing.

The network port check

The network port check would monitor a network port and report if it is responding.

```
define service {
    use                 generic-service
    hostgroup_name      switches
    service_description Port 443 status
    check_command       check_snmp!ifOperStatus.443!-r 1
}
```

This service check queries for the IOD `ifOperStatus.443`, in which the `.443` part indicates that we want to check the port 443. The second argument `-r 1` indicates that we expect the value `1` to be returned (`1` means it is in the UP state). So it will give:

- CRITICAL if SNMP returns a value other than `1`
- OK if SNMP returns the value `1`

Parent-child relationships and service dependencies

Icinga gives an ability to define the parent-child host relationships and service dependencies among service checks. This is important because:

- If a switch or router fails, all servers behind them will also become unreachable. Then Icinga would generate host alerts for *all* such servers and for the service checks defined on them. If we define the network device to be the parent of the servers behind it, Icinga will automatically suppress notifications for hosts behind the network device and perform service checks on them.

- If the SSH or NRPE service fails on a Linux or Windows server respectively, all service checks relying on these services would also fail and we will have many alerts. So we define service dependencies and make other services dependent upon SSH and NRPE. This would tell Icinga to suppress alerts for *dependent* service checks if the service *depended* upon fails.

Relationships between the hosts

Declaring relationships between the host is simple. The host object type provides the `parents` directive for this purpose. Suppose we have `server1.example.org` and `server2.example.org` behind a `sw-1.example.org` switch. When the switch goes down, Icinga will generate DOWN alerts for all servers behind it. In most cases, this is not desirable because once we get alerted about the switch being down, it is expected that the hosts behind it will also be unreachable. To suppress these subsequent alerts, we need to define relationships between the hosts in the host definitions as follows:

```
define host {
    use             generic-switch
    host_name       sw1.example.org
    address         192.168.32.1
    hostgroups      all,switches
}

define host {
    use             linux
    host_name       server1.example.org
    address         192.168.32.56
    hostgroups      all,linux
```

```
    parents        sw1.example.org
}

define host {
    use            windows
    host_name      server2.example.org
    address        192.168.32.57
    hostgroups     all,windows
    parents        sw1.example.org
}
```

The preceding definitions will make `sw1.example.org` the parent of `server1.example.org` and `server2.example.org`. When Icinga detects it cannot reach either of the servers, its reachability logic comes in and checks for reachability of their parents' and children's hosts and figures out the point of failure in the network map. It then marks the hosts behind the point of failure in an UNREACHABLE state.

By default Icinga will send notifications for an UNREACHABLE state too. So to suppress the notifications for such cases, we need to remove the unreachable (u) option from the value of the `notification_options` directive in host definitions.

```
define host {
    use                   generic-host
    host_name             server1.example.org
    notification_options  d,r
}
```

This can optionally be put in the host templates to affect all the hosts. Now with these two things (host relationships and exclude (u) option in host definition) configured, we will get only one notification when the switch is down.

Note that the web interface will still show all the servers and service checks on them as CRITICAL; only their notifications (e-mail and so on) will be suppressed.

Using these host relationships, Icinga builds a network map consisting of a graph with network devices and hosts at nodes connected from the monitoring servers to the root-level hosts in the tree, and then their children as their nodes. Following is a screenshot of an example network map:
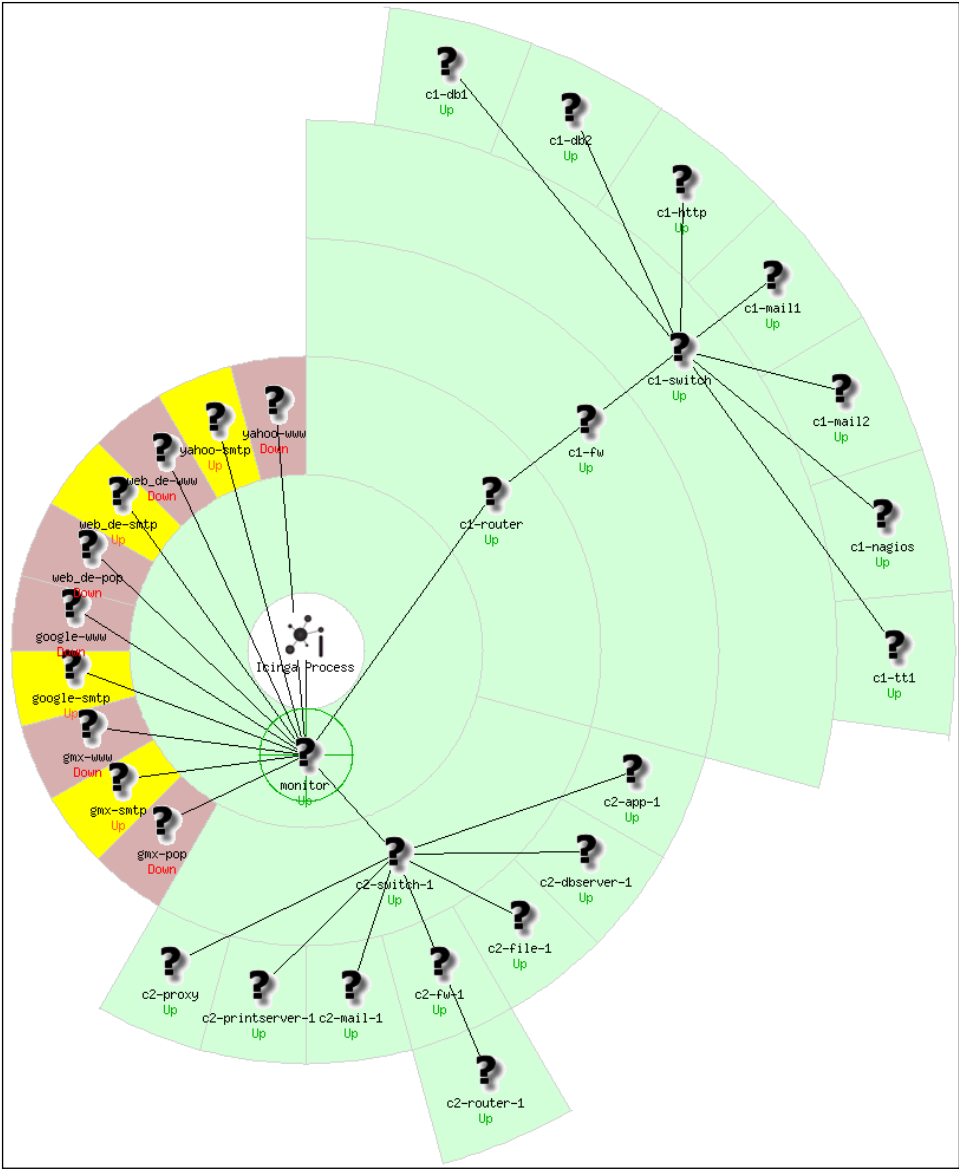
Network Map

Service relationships

Declaring service relationships is a little more complicated compared to host relationships. Here, we need to define Icinga objects of the type servicedependency to declare such dependency relationships.

Let's say we have a server.example.org server with the following host definition and ping, then HTTP and SMTP service checks:

```
define host {
    use                     generic-host
    host_name               server.example.org
}

define command {
    command_name            check_ping
    command_line            $USER1$/check_ping -H $HOSTADDRESS$
}
define service {
    use                     generic-service
    host_name               server.example.org
    service_description     Ping
    check_command           check_ping
}

define command {
    command_name            check_http
    command_line            $USER1$/check_http -H $HOSTADDESS$
}
define service {
    use                     generic-service
    host_name               server.example.org
    service_description     HTTP
    check_command           check_http
}

define command {
    command_name            check_smtp
    command_line            $USER1$/check_smtp -H $HOSTADDESS$
}
define service {
    use                     generic-service
    host_name               server.example.org
    service_description     SMTP
    check_command           check_smtp
}
```

With this configuration, we would want that when the ping check fails, we will know other checks would also fail, and there would be no unnecessary alert floods regarding them. So, we have to make HTTP and SMTP checks dependent on ping check. Here's how we see the definition:

```
define servicedependency {
    host_name                       server.example.org
    service_description             Ping
    dependent_service_description   HTTP,SMTP
}
```

The object definition would make the checks in the `dependent_service_description` directive depend on the checks in the `service_description` directive. Note that the values of these directives should match those specified in the `service_description` directive in service object definitions. Both the directives can have a comma-separated list of service descriptions, if needed. The following figure depicts the dependencies:
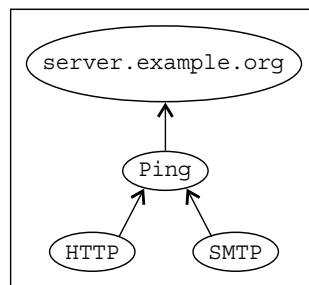
Dependency graph of Ping, HTTP, and SMTP service checks

Similarly for checks executing over SSH, we would define similar dependency objects. Let's look at the example of making load and disk checks dependent on the SSH check:

```
define servicedependency {
    host_name                       server.example.org
    service_description             SSH
    dependent_service_description   Load,Disk
}
```

The SSH check, of course, should in turn be dependent upon the ping check:

```
define servicedependency {
    host_name                       server.example.org
    service_description             Ping
    dependent_service_description   HTTP,SMTP,SSH
}
```

The following figure depicts this:
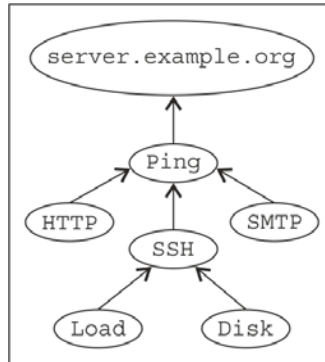
Dependency graph for Ping, HTTP, SMTP, SSH, load, and disk services

A similar dependency can be defined for NRPE checks on the Windows servers.

With these dependency relationships in place, Icinga now has a dependency tree-map of service checks which it would use to figure out what notifications it should suppress. Failing to perform the ping check won't alert about HTTP, SMTP, and SSH checks failing. SSH check failing won't alert about load and disk checks failing.

But this approach of configuration may be cumbersome since we have to do this separately for each host and dependent service check. This use case is a relatively common one that is used across most of the servers. In this case, we can pass hostgroup to service dependency.

```
define servicedependency {
    hostgroup_name                      linux
    service_description                 SSH
    dependent_service_description       Load,Disk
}

define servicedependency {
    hostgroup_name                      windows
    service_description                 NRPE
    dependent_service_description       Memory,Windows Folder Size
}
```

This would apply this dependency relationship on all hosts of the specified hostgroups. But this is still a little cumbersome since we have to keep adding service checks to the list in the `dependent_service_description` directive as and when they come up, which may get overlooked at times. So what we can do is define service groups (this is the same as hostgroups, that is, a group of *services*),put all dependent service checks in the service groups, and then simply specify the service group in the dependency object.

One could argue that it is one and the same because we will ultimately need to put a list of service checks in the service groups' definition. But that's not the only option, there's a better way to do it.

Let's define `ssh_dependent` service group, which will have relevant service checks as its members:

```
define servicegroup {
    servicegroup_name          ssh_dept
}

define service {
    ...
    service_description        Load
    ...
    servicegroups              ssh_dept
}

define service {
    ...
    service_description        Disk
    ...
    servicegroups              ssh_dept
}
```

Using the `servicegroups` directive in service definition, we can easily assign member service checks to desired service groups. Then, we only need to use this service group in the dependency definition:

```
define servicedependency {
    service_description             SSH
    dependent_servicegroup_name     ssh_dept
}
```

Note that the preceding definition assumes there is a SSH service check defined for servers having load and disk checks.

The preceding dependency definition would make all the service checks that are members of the ssh_dept service group depending on the SSH service check of respective hosts. Note that, in this method, we don't require a hostgroup to be specified in the dependency object definition. We have SSH and other service checks (disk, load, and so on) that are involved in the dependency defined to apply on the linux hostgroup. Icinga will apply this dependency on all hosts that have these service checks. Similar configuration can be applied to Windows servers:

```
define servicegroup {
    servicegroup_name           nrpe_dept
}
define service {
    ...
    service_description         Memory
    ...
    servicegroups               nrpe_dept
}
define service {
    ...
    service_description         Windows Folder Size
    ...
    servicegroups               nrpe_dept
}
define servicedependency {
    service_description             NRPE
    dependent_servicegroup_name     nrpe_dept
}
```

For common checks for public services, the configuration is as follows:

```
define servicegroup {
    servicegroup_name           ping_dept
}
define service {
    ...
    service_description         HTTP
    ...
    servicegroups               ping_dept
}
define service {
    ...
    service_description         SMTP
    ...
    servicegroups               ping_dept
}
define servicedependency {
    service_description             Ping
    dependent_servicegroup_name     ping_dept
}
```

The following figure shows the service group dependencies:
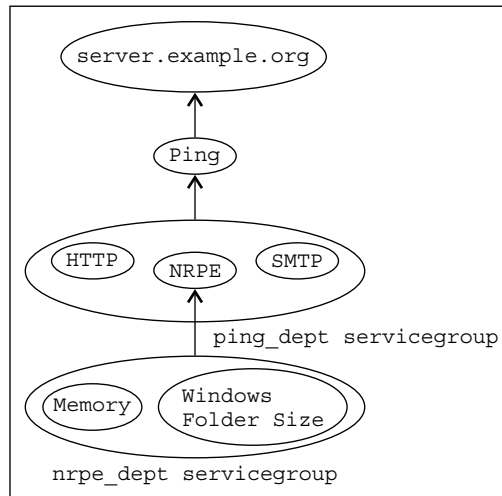
Service group dependency graph

Summary

This chapter covered the most commonly used system health checks that looked into most aspects of health monitoring for Windows/Linux servers and network devices, along with sample configuration and the best practices that should be followed.

We also saw how we can define relationships among hosts and dependencies among services to suppress alerts during a relatively big outage or a system failure and let Icinga pinpoint administrators to the point of failure in the infrastructure.

In the next chapter, we will move on to look at notification configuration and the options to customize notification options, and also how to write custom plugins.

5

Host and Service
Availability Reporting

Alerting is an important aspect of monitoring. There is no reason in having monitoring systems if we don't have efficient and impromptu alerting mechanisms in place. As discussed in previous chapters, Icinga provides a flexible way of configuring the type of alerts to send. Icinga itself isn't aware of the alerting mechanism. It simply knows when to call a notification (shell) command according to the Icinga notification configuration. Icing supplies the necessary details about the problem to the command, and the command takes over from there to use whatever mode of alerting it is programmed to use and actually send the alert. This flexibility allows us to easily extend and add custom alerting mechanisms, and simply configure Icinga to use it.

Each host/service definition has Icinga contact objects associated with it (using the `contacts` directive, a comma-separated list). Each contact object has directives to specify the name of the Icinga command object that it should use for notification. This command object has a directive to specify the exact command (as it would look in a terminal) to execute the command object. So, when a host/service check goes *critical*, its contacts are looked up and the associated command objects are called, which in turn execute the command that actually handles the sending of the notification.

This notification command can be a script to send an e-mail or an SMS, or to have a message posted to a Jabber contact (using a Jabber bot), or an API call (using cURL for instance) to an external system. The script is usually put at a convenient location, which is called by the notification command and is executed to do its processing.

In this chapter, we will understand how to configure Icinga for notifying contacts about problems, and customize various associated parameters.

Default configuration

The default Icinga installation has basic monitoring already setup (the one that we saw for localhost monitoring) along with e-mail alerting. Let's take a quick look again at one of the service checks, HTTP.

The service check definition, as in `localhost.cfg`, is as follows:

```
define service {
    use                    local-service
    host_name              localhost
    service_description    HTTP
    check_command          check_http
#   notifications_enabled  0       ; make sure this is commented
}
```

The HTTP check inherits the `local-service` template service, which in turn inherits the `generic-service` template service that has the contact information (`templates.cfg`):

```
define service {
    name                          generic-service
    active_checks_enabled         1
    passive_checks_enabled        1
    parallelize_check             1
    obsess_over_service           1
    check_freshness               0
    notifications_enabled         1
    event_handler_enabled         1
    flap_detection_enabled        1
    failure_prediction_enabled    1
    process_perf_data             1
    retain_status_information     1
    retain_nonstatus_information  1
    is_volatile                   0
    check_period                  24x7
    max_check_attempts            3
    normal_check_interval         10
    retry_check_interval          2
    contact_groups                admins
    notification_options          w,u,c,r
    notification_interval         60
    notification_period           24x7
    register                      0
}
```

The preceding template has `contact_groups` defined as `admins`, which is an Icinga `contactgroup` definition (essentially a group of Icinga contact objects) in `contacts.cfg`:

```
define contactgroup {
    contactgroup_name       admins
    alias                   Icinga Administrators
    members             icingaadmin
}
```

The `admins contact` group has `icingaadmin` as a contact member:

```
define contact {
    use             generic-contact
    contact_name    icingaadmin
    alias           Icinga Admin
    email           icinga@localhost
}
```

The `email` directive in the preceding definition should reflect the e-mail address that we had set in *Chapter 1*, *Installation and Configuration*, in order to receive a test alert.

So far, we have the HTTP service check associated with the `admins` contact group which has `icingaadmin` as a member contact, and which has an e-mail address. So far so good. We still don't have the notification command used to send alerts. Let's dig deeper.

The `icingaadmin` contact inherits a `generic-contact` template contact, which has the notification commands (`templates.cfg`):

```
define contact {
    name                            generic-contact
    host_notification_period        24x7
    service_notification_options    w,u,c,r,f,s
    host_notification_options       d,u,r,f,s
    service_notification_commands   notify-service-by-email
    host_notification_commands      notify-host-by-email
    register                        0
}
```

The `contact definition` template has both a `service_notification_commands` directive and a `host_notification_commands` directive that specify the notification commands to be used for service alert and host alert respectively. Since we have the HTTP service check into consideration for now, we will look at the former. The `notify-service-by-email` object is an Icinga command object (`commands.cfg`):

```
define command {
    command_name    notify-service-by-email
    command_line    /usr/bin/printf "%b" "***** Icinga *****\n\
nNotification Type: $NOTIFICATIONTYPE$\n\nService: $SERVICEDESC$\
nHost: $HOSTALIAS$\nAddress: $HOSTADDRESS$\nState: $SERVICESTATE$\n\
nDate/Time: $LONGDATETIME$\n\nAdditional Info:\n\n$SERVICEOUTPUT$\n"
| /bin/mail -s "** $NOTIFICATIONTYPE$ Service Alert:
$HOSTALIAS$/$SERVICEDESC$ is $SERVICESTATE$ **" $CONTACTEMAIL$
}
```

This command object definition specifies the long `command_line` object to be executed for this command. The command line, in this case, builds the entire body of the e-mail, pipes it to `/bin/mail` that is passed a subject, and sends the e-mail to the address given by `$CONTACTEMAIL$`. The following figure illustrates the configuration objects and how they are related:
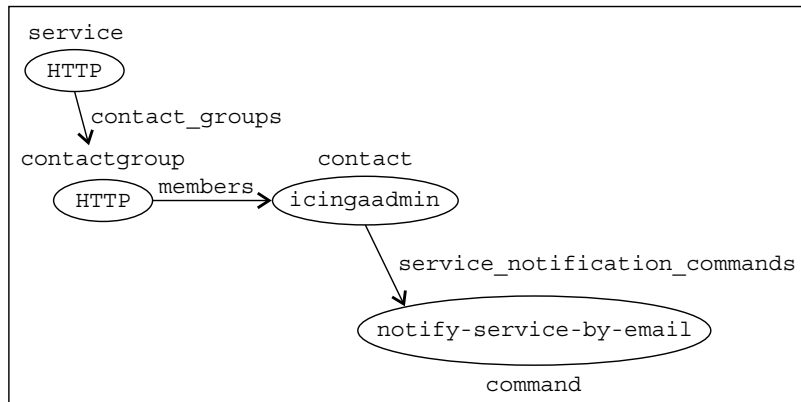
Relationship among service, contact group, contact, and command objects that form the notification configuration

As you may have noticed, there are a lot of parameters used in the command line. These are the Icinga macros that can be used inside object definitions. The Icinga documentation has an exhaustive list of available macros (and the types of objects each is available in). Let's look at the ones used here, with the values they will take for the HTTP service check example:

- `$NOTIFICATIONTYPE$`: This macro takes the type of notification to be sent, that is, problem/recovery. When Icinga detects the service to be CRITICAL, it sends a problem notification, and when it becomes OK, it sends a *recovery* notification.

- `$SERVICEDESC$`: This macro takes the value of the `service_description` directive in the service object for the check this notification is for. Our example will take the HTTP value.

- `$HOSTALIAS$`: This macro takes the value of the `alias` directive in host object to which this service check belongs. If this directive is not specified, it takes the value of the `host_name` directive. Our example will have a localhost as its value.

- `$HOSTADDRESS$`: This macro takes the value of the `host_address` directive in the same host object as above. Our example will take `127.0.0.1` as the value.

- `$SERVICESTATE$`: This macro takes the current state of the service check, that is, either one from CRITICAL, WARNING, UNKNOWN, or OK. When the HTTP service is down, this value will be CRITICAL, and when it recovers, the value will be OK.

- `$LONGDATETIME$`: This macro gives the date/time when the service check went from CRITICAL to OK. The format is defined with the `date_format` directive in the main `icinga.cfg` configuration file.

- `$SERVICEOUTPUT$`: This macro gives the output of the service check as reported by the check plugin. Our example will show something like HTTP CRITICAL, which means that the macro is unable to open the TCP socket as the value when the check goes CRITICAL.

- `$CONTACTEMAIL$`: This macro takes the value of the `email` directive in contact definition, in which the said command object is defined. Our example will take the e-mail address that we had defined in *Chapter 1*, *Installation and Configuration*, (`your@email.com`) as the value of this macro.

Each of the preceding macros is replaced with its value in the command line and the resulting command string is executed. The resulting command string for our example will look similar to the following:

```
/usr/bin/printf "%b" "***** Icinga *****\n\nNotification Type: PROBLEM\n\
nService: HTTP\nHost: localhost\nAddress: 127.0.0.1\nState: CRITICAL\n\
nDate/Time: 07-03-2013 13:47:52\n\nAdditional Info:\n\nHTTP CRITICAL -
Unable to open TCP socket\n" | /bin/mail -s "** PROBLEM Service Alert:
localhost/HTTP is CRITICAL **" your@email.org
```

The e-mail message, as we would receive it, would look similar to the following:

```
From: icinga@localhost

To: your@email.com

Subject: ** PROBLEM Service Alert: localhost/HTTP is CRITICAL **


***** Icinga *****


Notification Type: PROBLEM


Service: HTTP

Host: localhost

Address: 127.0.0.1

State: CRITICAL


Date/Time: 07-03-2013 13:47:52


Additional Info:


HTTP CRITICAL - Unable to open TCP socket
```

We will also receive a similar e-mail when the service recovers.

Customizing notification behavior

We looked at the basic and default notification configuration provided by Icinga. There are a number of customizations possible on top of it, some of which are covered in this section.

Service definitions

Service objects offer some directives to control or specify whether, which, and how the notifications should be sent out.

- The `notifications_enabled` directive is used to enable/disable (`0`/`1`) notifications for a service check. This is useful to perform extremely noncritical checks, for which we don't need notifications. The default value is `1`, and should only be set to `0` to disable notifications.

- The `notification_options` directive determines what kind of notifications should be sent for the service check. The notification options that can be specified as the value are as follows:

  - `w` = WARNING
  - `u` = UNKNOWN
  - `c` = CRITICAL
  - `r` = RECOVERY

  So, suppose we don't want to receive the WARNING and UNKNOWN notifications, we can simply have `c` and `r` as the value of this directive in the service definitions, which will send only the CRITICAL and RECOVERY notifications.

- The `first_notification_delay` directive determines the time to wait before sending out the first notification after a service check enters a non-OK state. The value is the number of time units, and the length of a time unit is defined by the `interval_length` directive in `icinga.cfg`, which defaults to 60 seconds. This is useful if we expect intermittent problems and that they will recover within a certain time automatically; we don't need a notification if it actually recovers within the expected time. The default value is `0`, which means Icinga should send out a notification immediately.

- The `notification_interval` directive determines the time after which a contact should be notified again that the service is still in the non-OK state. The value is again the number of time units. The service template has this set to `60`; it means that the contacts will be renotified about the problem if a service check stays in the non-OK state for more than 60 minutes (or 1 hour). To disable such reminder notifications, set this value to `0`.

- The notification_period directive describes the time of the day/week/ month/year to which the notifications should be restricted. This is specified by the Icinga timeperiod object. This timeperiod object defines the periods of time, such as selected days of a week, or selected hours in a day. The service template has the timeperiod object as 24x7, as the value of this directive. Let's look at its definition (timeperiods.cfg):

```
define timeperiod {
    timeperiod_name 24x7
    alias           24 Hours A Day, 7 Days A Week
    sunday          00:00-24:00
    monday          00:00-24:00
    tuesday         00:00-24:00
    wednesday       00:00-24:00
    thursday        00:00-24:00
    friday          00:00-24:00
    saturday        00:00-24:00
}
```

The preceding timeperiod object defines an all-time time period for all days in a week and all hours in a day. This means the notifications can be sent out for the said service at all times.

Similarly, we can restrict the time at which the alert for the service should be sent by defining a more restrictive timeperiod object and using this notification_period directive in the service definition. There is an timeperiod object named as workhours, already defined in default configuration; let's have a quick look at it for clarity:

```
define timeperiod {
    timeperiod_name workhours
    alias           Normal Work Hours
    monday          09:00-17:00
    tuesday         09:00-17:00
    wednesday       09:00-17:00
    thursday        09:00-17:00
    friday          09:00-17:00
}
```

The preceding timeperiod object defines normal work hours, that is, only 9 a.m. to 5 p.m. on weekdays. If this time period is used in any of the service checks, no notifications will be sent for the service checks outside of these work hours.

Contact definitions

Notification options can also be customized inside contact definitions. Let's look at various available directives:

- The `host_notifications_enabled` and `service_notifications_enabled` directives can be used to enable/disable (`0`/`1`) the host/service notifications on the basis of contacts. The default value is `1`. The value(s) should be set to `0` to explicitly disable notifications for the contact.

- The `host_notification_period` and `service_notification_period` directives can, again, be used to specify the Icinga `timeperiod` objects, only within which the notifications to this particular contact should be sent out. This may be useful if only an on-duty person should receive notifications 24x7 and the boss should receive it only during work hours.

- We have already looked at `host_notification_commands` and `service_notification_commands`. Note that the value of these directives can be a comma-separated list of command names if we want multiple commands to be executed for notifications.

- The `email` directive, as seen in the earlier sections, is used to specify the e-mail address of the contact. This is accessible using the `$CONTACTEMAIL$` Icinga macro in command objects.

- The `pager` directive is used to specify the mobile number of the contact, which is available via the `$CONTACTPAGER$` Icinga macro in command objects.

- The `addressx` directives are used to specify the other miscellaneous addresses for the contact (Jabber ID and so on). The value of `x` ranges from `1` through `6`, each of which is available via the `$CONTACTADDRESSn$` macros, where again `n` ranges from `1` through `6`.

The host/service escalation

Escalation is basically if `person1` receives an alert and the problem continues to be there for `x` number of minutes then a notification should be sent (escalated) to `person2`.

Escalation is another aspect of notification configuration and can be done using the `hostescalation` and `serviceescalation` Icinga objects. These objects are used to define escalation paths to various people. This path can go on as long as we have configured it.

The escalations work when there is `notification_interval` defined in the service definition, which will re-notify contacts after the defined interval. Escalation logic kicks in when the `serviceescalation` object is defined. In the object definition, we define the contacts that should be notified at the nth renotification.

For example, we have three contacts defined: `onduty`, `techlead`, and `manager`. We want the `onduty` contact to be notified immediately, `techlead` if the check does not recover within 15 minutes, then notify the `techlead` if it does not recover within the next 15 minutes (30 minutes from the start), and finally to notify the `manager` contact if it does not recover within the next 15 minutes (45 minutes from start). For such a scenario, we need to set the `notification_interval` directive to 15 minutes (assuming the length of a time unit is one minute in the main configuration) in a service object definition, so that Icinga will retrigger the notification every 15 minutes until the check recovers. Now, we also need to define two `serviceescalation` objects as follows (we will use the `localhost` host and the HTTP service as examples; this can be used in the same way for remote hosts too):

```
define serviceescalation {
    host_name               localhost
    service_description     HTTP
    first_notification      2
    last_notification       3
    contacts                techlead
}

define serviceescalation {
    host_name               localhost
    service_description     HTTP
    first_notification      4
    last_notification       0
    contacts                manager
}
```

The preceding escalation definitions have the `first_notification` and `last_notification` directives, which determine the number of notifications for which the given escalation is valid. The `first_notification` directive specifies n for *n*th notification, with which this escalation becomes valid. The `last_notification` directive specifies n for *n*th notification, with which the escalation becomes ineffective. Escalation being valid implies that the contacts specified in the escalation will be notified. When the service goes CRITICAL, the `onduty` contact is immediately notified as the first notification.

For the first escalation definition, escalation becomes valid with the second notification (`first_notification` is 2) for the service and will become ineffective after the third notification (`last_notification` is 3) has been sent out. So, the second and the third notifications will be sent to the `techlead` contact after 15 and 30 minutes each.

For the second escalation definition, escalation becomes valid with the fourth notification (`first_notification` is 4) being sent out for the service and will never become ineffective (`last_notification` is 0). So the fourth notification (after 45 minutes) will go to the `manager` contact and will be notified every 15 minutes until the service check recovers.

The following table puts together a timeline for this example:

Timestamp	Service State	n (nth notification)	Explanation
05:00	CRITICAL	1	The `onduty` contact is immediately notified.
05:15	CRITICAL	2	First `serviceescalation` kicks in (`first_notification` is 2) and the `techlead` contact is notified.
05:30	CRITICAL	3	The `techlead` contact is notified again and first `serviceescalation` ends (`last_notification` is 3).
05:45 and onward	CRITICAL	4	Second `serviceescalation` kicks in (`first_notification` is 4), the `manager` contact is notified and will be notified every 15 minutes until the service recovers, as this escalation never ends (`last_notification` is 0).

A similar configuration can be made for host escalations too:

```
define hostescalation {
    host_name               localhost
    first_notification      2
    last_notification       3
    contacts                techlead
}

define hostescalation {
    host_name               localhost
    first_notification      4
    last_notification       0
    contacts                manager
}
```

Let's take a quick look at various other directives offered by escalation objects to control notifications:

- The `first_notification` and `last_notification` directives, as we saw earlier, define the number of notifications for which the escalation stays valid.

- The `notification_interval` directive can be used to specify the time to wait before sending a notification to the contact in the escalation that is similar to the directive in service definition.

- The `escalation_period` directive can be used to specify the time period object for which the escalation stays valid (apart from the first and last notification numbers). The same `timeperiod` object can be used, and is optional.

- The `escalation_options` directive is used to specify which notifications to send to the contacts of this escalation. Options are as follows:

  ○ `w` = WARNING

  ○ `u` = UNKNOWN

  ○ `c` = CRITICAL

  ○ `r` = RECOVERY

  This is similar to the corresponding directive in the service definition.

- The `first_warning_notification`, `last_warning_notification`, `first_critical_notification`, `last_critical_notification`, `first_unknown_notification`, and `last_unknown_notification` directives are used to specify different notification numbers for the WARNING, CRITICAL, and UNKNOWN states of the service check. This works in a similar way to the first and last notification number directives, except that the one that is more applicable according to the service state is used.

This covers most of the configuration options that Icinga provides to customize the behavior of notifications.

Summary

At this point, we have seen how basic notifications work, starting from the service check, to contact, to command. We saw how to restrict notifications to be sent only at a certain time, how to send a specific type of notifications. We also saw how to configure various contacts to receive notifications if a long duration outage occurs without manual intervention from people already troubleshooting the issue and relevant people need to be notified. A more comprehensive and minutely detailed explanation of various directives can be found in Icinga's official object definition documentation.

In the next chapter, we look into how we can extend Icinga with custom plugins and leverage its flexibility to suit our customization requirements.

6
Icinga Plugins

As mentioned in earlier chapters, Icinga maintains abstraction over the internal implementation of check plugins and how the status checks are performed. It only knows what it has to monitor and what commands it has to use to get the status of the services that are being monitored. It does not have details about any internal mechanism for performing status checks; it relies on external programs and commands to do the dirty work.

These external programs are called **plugins**. A plugin can be any executable (a compiled program or a script), which can be run from the command line. Icinga will execute the plugin configured for each service check and determine the status of the service from the exit status of the plugin. Icinga is not aware of the internal logic or implementation of the plugin itself. Once Icinga has the status of the service check, it calls notification commands or event handlers based on its configuration.

Note that the plugins can also be used to perform host checks. Each host object definition also has a `check_command` directive, which refers to the Icinga command object that is used to determine the status of the host. This check command can refer to a plugin that implements the checking of the host status and reports the status back to Icinga. So depending on whether the plugin is used for a host status check or service status check, it sets the relevant macros to values as reported by the plugins. This will be explained shortly.

Icinga's default installation comes with a number of plugins, and a few others are provided by the `nagios-plugins` package. Other plugins for performing more specific checks are available as separate packages, for example, `nagios-plugins-dhcp`. Most standard check plugins provide a `--help` command-line option to see the usage and options provided by the plugin.

Writing custom plugins

It is also possible to write custom check plugins, which can be scripts (bash, python, and so on) or compiled programs (C, Java, and so on). These plugins need to comply with a minimum two things:

- The exit codes, `0=OK`, `1=WARNING`, `2=CRITICAL`, and `3=UNKNOWN`, which are stored in the `$HOSTSTATEID$` and `$HOSTSTATE$` macros for host checks, and the `$SERVICESTATEID$` and `$SERVICESTATE$` macros for service checks

- At least one line of output written to `STDOUT`, which is stored in the `$HOSTOUTPUT$` macro for host checks and the `$SERVICEOUTPUT$` macro for service checks

Apart from these, the plugin can output the following:

- Multiline output written to `STDOUT`; Icinga will accept and store more lines of output in the `$LONGHOSTOUTPUT$` or `$LONGSERVICEOUTPUT$` macro.

- Output written to `STDOUT` is stored as performance data for other applications, which need to be processed. It is stored in the `$HOSTPERFDATA$` or `$SERVICEPERFDATA$` macro. (The performance data is used for various graphing and reporting purposes among other possible use cases.)

  The format of the output that will be parsed into these macros is as follows:
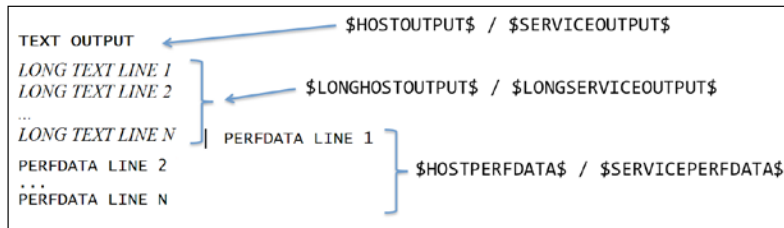
Plugin output interpretation (The fonts of the lines are changed for representational purpose only)

The line seen in the bold style is stored in the `$HOSTOUTPUT$` or `$SERVICEOUTPUT$` macro and the rest of the parts are optional. The lines seen in the italic style are stored in the `$LONGHOSTOUTPUT$` or `$LONGSERVICEOUTPUT$` macro. The lines in normal text format are stored in the `$HOSTPERFSDATA$` or `$SERVICEPERFDATA$` macro. The lines of output and performance data output are separated by pipe (|) symbol.

For example, if we have a check script that writes the following output to STDOUT:

Sample output of the check_disk check plugin

The bold part of the output will be stored in the $SERVICEOUTPUT$ macro and the part seen in normal font will be stored in the $SERVICEPERFDATA$ macro. Since there are no long text lines between first line of output and the pipe symbol, the $LONGSERVICEOUTPUT$ macro for the long output will have an empty value.

Another example of a check script output is as follows:

Full sample output of the check_disk check plugin

Icinga will only read the first 8 KB of output that a plugin writes to STDOUT. This is to prevent overflow of data reported to Icinga.

> All plugins that the Icinga server executes are executed as the operating system user that is configured in the main configuration file icinga.cfg by the value of the icinga_user directive, which is icinga by default. We have to make sure to have proper permissions in place (make it executable for the icinga user, give sudo-permissions permissions if required, and so on).

Integrating custom plugins

Let's say we wrote a custom `sample-plugin.sh` check plugin that takes some arguments and gives check results.

First, we need to put the check plugin at a proper path in the filesystem. Place the check script inside the standard plugin installation directory. RedHat/CentOS distributions install the plugins inside `/usr/lib64/nagios/plugins` or `/usr/lib/nagios/plugins`, depending on the architecture of the server; this may differ for other distributions. Place the check script in one of these directories and make sure it is executables the operating system user that Icinga is running as.

Check if we can run the script from the command line as an `icinga` user:

```
$ su icinga -c '/usr/lib64/nagios/plugins/sample-plugin.sh 192.168.1.212'

Sample check OK
```

We use the following command with the path in the Icinga command object definition; let's call it `check_sample`:

```
define command {

    command_name    check_sample

    command_line    /usr/lib64/nagios/plugins/sample-plugin.sh
$HOSTADDRESS$ $ARG1$ $ARG2$

}
```

Now we can use the following command object in service definitions using the `check_command` directive:

```
define service {

    ...

    check_command    check_sample!argument1!argument2

    ...

}
```

As mentioned earlier, arguments to be passed to the check script are separated by an exclamation (`!`) sign in the `check_command` directive in a service definition. The preceding service check will expand the command line into the following (assuming the service check is on a host with the address `192.168.1.212`):

```
/usr/lib64/nagios/plugins/sample-plugin.sh 192.168.1.212 argument1
argument2
```

The `$ARGn$` macros are defined by corresponding arguments (separated by `!`) passed to the `check_command` directive.

Threshold and range standards

Most checks are based on some definite range of threshold values. For example, generate critical alert if the CPU load is more than 80 percent, warning alert if it is between 60 and 80 percent. These values are not restricted to percentages, they can be virtually any numbers depending on our check's logic and requirements.

Most check plugins follow a convention in taking threshold ranges as command-line arguments. This convention may not be followed by all plugins, but most of the standard ones do. This is a recommended way of taking threshold ranges so that it is easy to understand the usage of the plugin by other potential users. The ranges are generally in following format:

```
[@]start:end
```

Things to keep in mind about various usages and corresponding interpretations of the following:

- Range is indicated by a `start` value and an `end` value, and is inclusive of both.
- Only `start` means `end` is infinity; only `end` means `start` is zero (note that the latter does not need a colon (`:`)).
- Alert is raised if value is outside the range. If `@` is used, alert is raised if value is inside the range.

Let's look at some examples of ranges:

Range example	Generate alert if value is
10:20	<10 or >20 (outside range {10..20})
10:	<10 (outside range {10..inf})
10	>10 (outside range {0..10})
@10:20	>=10 and <=20 (inside range {10..20})

These threshold values/ranges are usually used for generating warning and critical alerts. The `-w` command-line switch is used to specify a threshold value/range for the warning alert, and the `-c` switch is used to specify the threshold value/range for the critical alert. This, again, is a convention, many plugins may not follow.

And some command-line examples are given in the following table:

Command line	Interpretation
check_sample -w10 -c20	Critical if >20, warning if >10
check_sample -w10: -c20	Critical if >20, warning if <10
check_sample -w20:30 -c10:40	Critical if <10 or >40, warning if <20 or >30

This is just a convention for using the plugins. Many plugins don't follow it, and it does not apply to plugins that don't need threshold ranges.

Summary

Check plugins (also known as scripts/programs) implement the actual logic for checking the status of the desired service; Icinga knows nothing about this logic. It only knows about the exit code of the plugin and one or more lines of text written by the plugin to give a standard output. Both of these are used to determine and describe, respectively, the status of the service in question.

Once the service check status is known, relevant notification logic is used to notify contacts whenever there is a change in the state. The alert message is constructed using the macros whose values are set according to the exit code and output of the plugins.

There are few standards to writing plugins, including the proper exit code and output text as understood by Icinga, and accepting warning/critical threshold ranges/values. The former is required, while the latter is optional.

In the next chapter, we will take a look at some available web interfaces for an Icinga monitoring server. They can be used to view service check statuses of all services in a couple of different forms. We will look into how to configure and deploy web interfaces and will give an overview of the user-side interface components.

7

Web Interfaces

In the earlier chapters, we looked into most of the important configuration parameters that Icinga provides. We can use them to set up the monitoring configuration best suited for our requirements. We now proceed to look into the available web interfaces that are actually used to access the information and current status of our infrastructure to perform required actions.

Icinga comes with a default web interface, known as Icinga Classic. It is the most basic one in terms of the user interface, providing all available ways to access information and perform action. It also has another more modern web interface, known as Icinga Web. It is advanced in terms of usability, and uses more of AJAX to make the interface intuitive. Both versions have a similar set of features.

There are many independent web UIs and the Thruk project (`www.thruk.org`) is one of them. It supports a number of monitoring backends such as Nagios, Icinga, and Shinken. It is again more intuitive in usability, but is similar to the classic interface in design. Thruk has the ability to connect to multiple instances of monitoring backends and show the consolidated information in one interface. All the classic web interfaces lack this functionality.

Both Icinga Classic and Thruk use CGI scripts to access information from the Icinga core and show it in the browser. Each CGI script corresponds to a view in the interface. We will look at each of these views and get ourselves acquainted with what the interface provides, and what we can do with it.

Icinga Classic

Icinga Classic is the default web interface for Icinga. In this section, we will look into various views that this interface provides.

Authentication

Icinga's Apache configuration, by default, requires the user to provide an authentication to access the web interface. This is a simple HTTP authentication using a `htpasswd` file to store usernames and encrypted passwords. The default username and password is `icingaadmin` and `icingaadmin` respectively. You should change this using the `htpasswd` utility. It can be used to set/update passwords and add new users.

```
# create johndoe user or update its password:
htpasswd /etc/icinga/htpasswd johndoe
```

If the username is the name of a valid Icinga contact, the web interface will show only those hosts and services that are associated with this contact, either directly by contacts directive or indirectly via contact groups. So, operational teams managing different set of servers can have a contact associated with relevant hosts and services, set a password for that username in the `htpasswd` file, and use the authentication details to see only relevant information about the hosts and services.

The Status view

The **Status** view shows a quick overview of the on-going problems in the network. Here is a screenshot of the view from Icinga's official demo website:

The status view

The current default view is the **Service Problems** page under the **Problems** section. In the preceding screenshot, the table in the center shows that the hosts are **gmx-pop**, **gpx-www**, **web_de-pop**, and so on, all these hosts are unreachable, and the relevant services on them are also critical. Some services such as SMTP on **gmx-smtp** are critical although the servers are still reachable. The table shows short information on details such as the time when last check was performed, the duration that the service has been critical for, and the output of the check plugin describing the problem.

The top panel shows a statistical overview of the current network status. The first row on the left-hand side shows the number of hosts in various states. The second row shows the numbers for services. The three numbers in each state correspond to unacknowledged, acknowledged, and total problems in that state. The last entry **TOTAL** in each row shows the number of problems and number of all hosts and services. On the top-right corner (inside the panel), the statistics about the Icinga core process running are available. You will see indicators for minimum/maximum service check execution time, latency, and so on.

The area above the table in the center gives filtering and pagination options. There is also a commands dropdown on the right-hand side, providing commands such as schedule downtime, and acknowledge a problem. Icinga supports some external commands that include setting downtime for service checks during a migration, acknowledging a problem to prevent further notifications, or for the purpose of records.

Users can select multiple problems at once using the checkbox at the end of each row in the table, and execute the selected command on all of them at once. This is helpful if there is a widespread outage and we need to acknowledge a number of problems on the screen at once.

On the left-hand side of the screen, there is a navigation bar for quickly accessing relevant information in the center part.

A tactical overview

The tactical view is the default view that opens when we open up the web interface. It is under the **Status** category, as shown in the following screenshot:
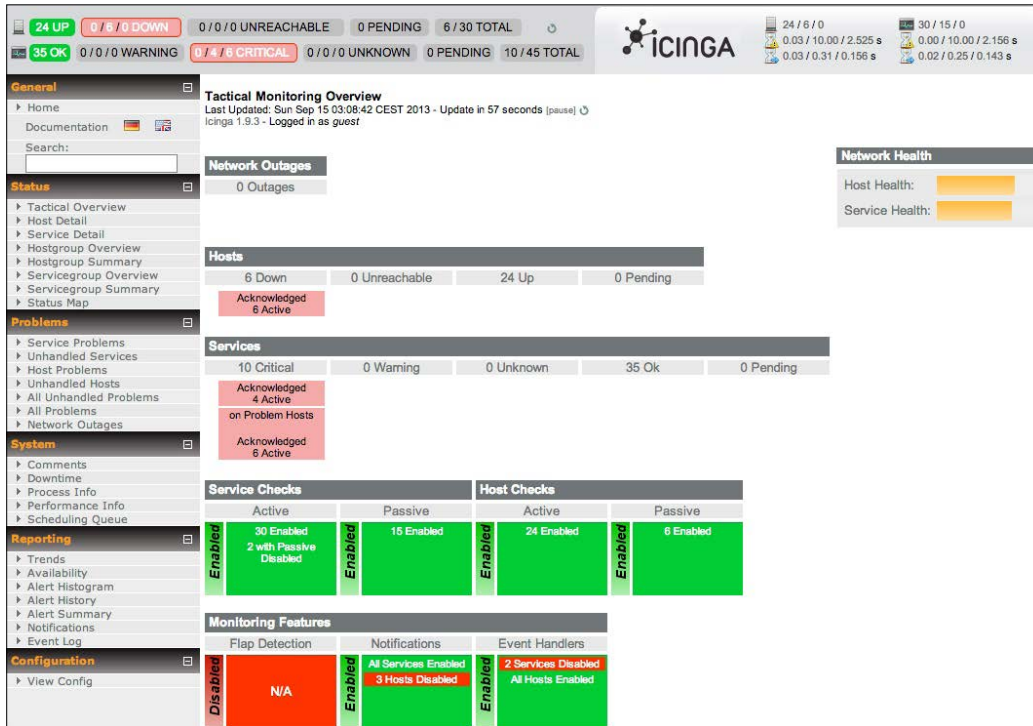
A tactical overview

This view gives an overview of the network health in terms of host and service health. It shows the number of acknowledged and unacknowledged problems in various states, along with information such as active/passive checks, flap detection configuration, notifications being sent, and event handlers being called.

The panel at the top and navigation bar on the left-hand side remains the same in all the views; however, the central part keeps changing.

The host/service detail

The **Host Detail** and **Service Detail** entries are under the **Status** section in the navigation bar.

Service details

Each of these views shows the list of all hosts/services and their status. The default view shows only those hosts and services that are in one of the problem states. This helps when setting up the monitoring configuration, and need to make sure that all hosts and services are in place.

The Hostgroup/Servicegroup Overview/ Summary status

The **Hostgroup Overview**, **Hostgroup Summary**, **Servicegroup Overview**, and **Servicegroup Summary** entries are under the **Status** section. The overview page shows the hosts and services that are grouped by hostgroups/servicegroups, while the summary page shows numerical statistics for the same.

The status map

The status map view is an informative one, it shows the map of the entire network using information from the host-parent relationships, and it helps in analyzing the point of failure in case of an outage. Following is an example screenshot:
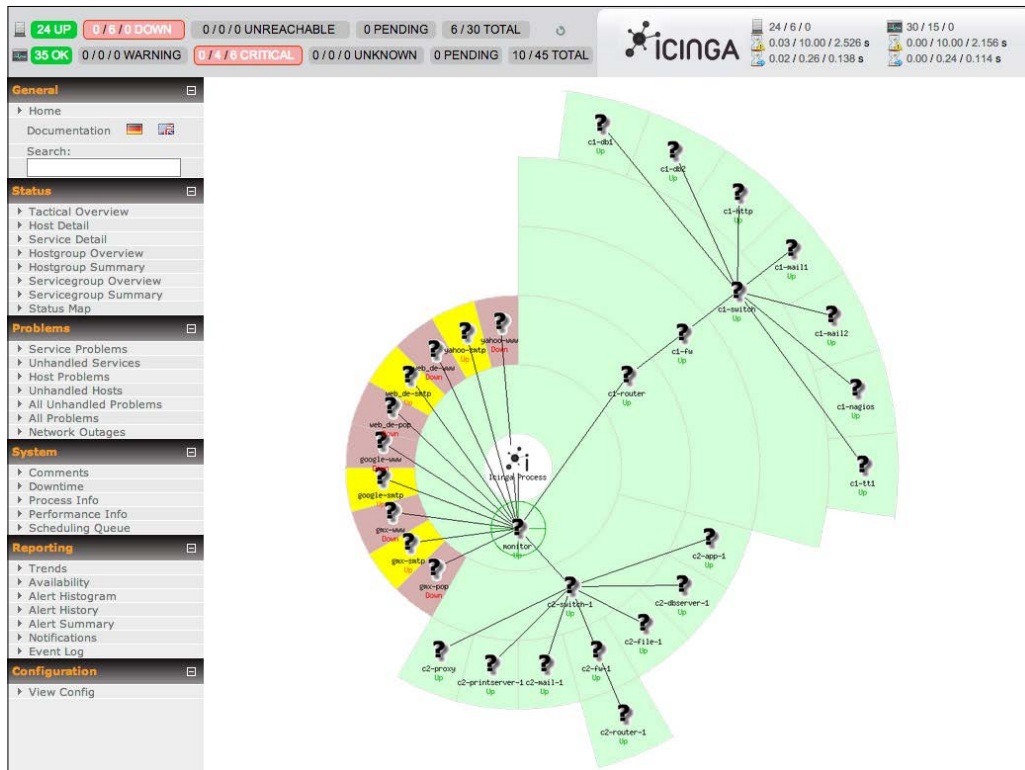
The status map

The preceding figure shows the network map in a graph-like form connecting routers/switches and servers. As we can see, the end points in red and yellow are in a problem state. They are directly connected to the Icinga process node as they don't have a proper host-parent (network device) set in their configurations.

The All Problems view

The **All Problem** view is a view under the **Problems** section. It is a clean view that shows separate lists of hosts and services in the problem state.

The All Problems view

Other views

The web interface comes with many other views to look into various things, such as the comments view that lists all comments and acknowledgements made by users on various checks, a downtime view to look at all hosts/services with scheduled downtime, and various other views to generate different reports.

Icinga Web

Icinga Web is developed as a separate subproject under the Icinga project umbrella. It is not included in the default Icinga package, hence it needs to be separately installed.

Requirements

Icinga Web requires a RDBMS backend such as MySQL, PostgreSQL, and Oracle. We will restrict ourselves to MySQL. You can refer to official installation guide for instructions for the other backends. Make sure you have a working MySQL server installation ready for use.

Install the following dependencies required for Icinga Web:

- Debian/Ubuntu:

  ```
  sudo apt-get install php5 php5-cli php-pear php5-xmlrpc php5-xsl
  php5-pdo php5-soap php5-gd php5-ldap php5-mysql
  ```

- RedHat/CentOS:

  ```
  sudo yum install php php-cli php-pear php-xmlrpc php-xsl php-pdo
  php-soap php-gd php-ldap php-mysql
  ```

Installation

Now we need to install the `icinga-idoutils-libdbi-mysql` and `icinga-web` packages. There are three ways to install them, follow the one you used for original Icinga installation:

- Upstream: We need to install the `icinga-idoutils-libdbi-mysql` and `icinga-web` packages. They should be available in your distribution packages. Similar to the `icinga` package, Debian (Squeeze/Wheezy), and Ubuntu have the upstream packages available on LaunchPad. RedHat/CentOS have it available in RepoForge YUM repository (`http://repoforge.org/`). Install these packages from the relevant source and skip to the *Configuration* section.

- Build the RPM packages: If you built the `icinga` package yourself (for example, for RedHat/CentOS), the `icinga-idoutils-libdbi-mysql` package is built along with it. Simply install it and continue with building the `icinga-web` RPM package by following these steps:

  1. Download the source. The source tarball can be downloaded from Icinga's official website from its **Downloads** section (`https://www.icinga.org/download/packages/`). The source includes a SPEC file for building RPMs.

  2. Build the RPM using the following command:

     ```
     $ rpmdev-setuptree
     $ cp icinga-web.spec ~/rpmbuild/SPECS
     $ cp icinga-web-*.tar.gz ~/rpmbuild/SOURCES
     $ cd ~/rpmbuild && rpmbuild -bb SPECS/icinga-web.spec
     ```

     This will create an `icinga-web` RPM in the `~/rpmbuild/RPMS/noarch` directory. Install it with the `rpm -ivh` command. Now continue with the *Configuration* section for configuration.

- Compiling from the source: If you installed Icinga from the source, you should already have `icinga-idoutils-libdbi-mysql` installed. Compile **icinga-web using the following commands:**

  $ tar zxvf icinga-web-1.9.1.tar.gz

  $ cd icinga-web-1.9.1

  $ ./configure

  $ sudo make install install-apache-config

> Refer to the official documentation in case of any problems (`http://docs.icinga.org/latest/en/icinga-web-scratch.html#install`).

Configuration

In this section, we will see how to configure the Icinga Web. Start MySQL Server using the following command:

$ sudo service mysqld start

IDOUtils

Icinga needs access to MySQL database server, and for that we need to initialize a new database with a schema provided by Icinga. If you're using a different database server than `localhost`, edit the following script to properly set the database host in the `DBHOST` variable, and run it on the command line:

**$ sudo /usr/share/doc/icinga-idoutils-libdbi-mysql-1.9.1/db/scripts/
create_mysqldb.sh**

Also, update the database configuration in `/etc/icinga/ido2db.cfg` in case the database is not on `localhost`. The path may vary depending on the distribution and version of the package installed. This script will initialize the database with the required schema. Now start the `ido2db` service:

$ sudo service ido2db start

Icinga Web

Icinga Web uses a separate database than to IDOUtils, hence we need to set up a separate MySQL user and database by following these steps:

1. Create a user by using the following commands:

   $ mysql -u root -p

```
 mysql> GRANT USAGE ON *.* TO 'icinga_web'@'localhost' IDENTIFIED
BY 'icinga_web';

  GRANT SELECT, INSERT, UPDATE, DELETE, CREATE, DROP, ALTER, INDEX
ON icinga_web.* TO 'icinga_web'@'localhost';

  quit
```

2. Create a database using the following command:

```
$ mysql -u root -p

 mysql> CREATE DATABASE icinga_web;

   GRANT USAGE ON *.* TO 'icinga_web'@'localhost' IDENTIFIED BY
'icinga_web' WITH MAX_QUERIES_PER_HOUR 0 MAX_CONNECTIONS_PER_HOUR
0 MAX_UPDATES_PER_HOUR 0;

   GRANT SELECT, INSERT, UPDATE, DELETE, CREATE, DROP, ALTER,
INDEX ON icinga_web.* TO 'icinga_web'@'localhost';

   FLUSH PRIVILEGES;

   quit
```

3. Initialize the database with the required schema as follows:

```
$ mysql -u root -p icinga_web < /usr/share/doc/icinga-web-1.9.1/
schema/mysql.sql
```

4. Replace `localhost` in the preceding commands with the hostname of the database server if it is different from the Icinga server. The default configuration of Icinga Web connects to MySQL at `localhost`, if this is not the case for your setup, update relevant entries in `/usr/share/icinga-web/app/config/databases.xml`.

5. Restart the web server and Icinga:

```
$ sudo service httpd restart
```
```
$ sudo service icinga restart
```

You can now access the Icinga Web at `http://localhost/icinga-web/` with username `root` and password `password`.

Screenshots

Icinga Web is more or less the same as the Icinga Classic interface in terms of the feature set. Let us have a quick look at some of the views, the screenshots have been taken from the official Icinga website.
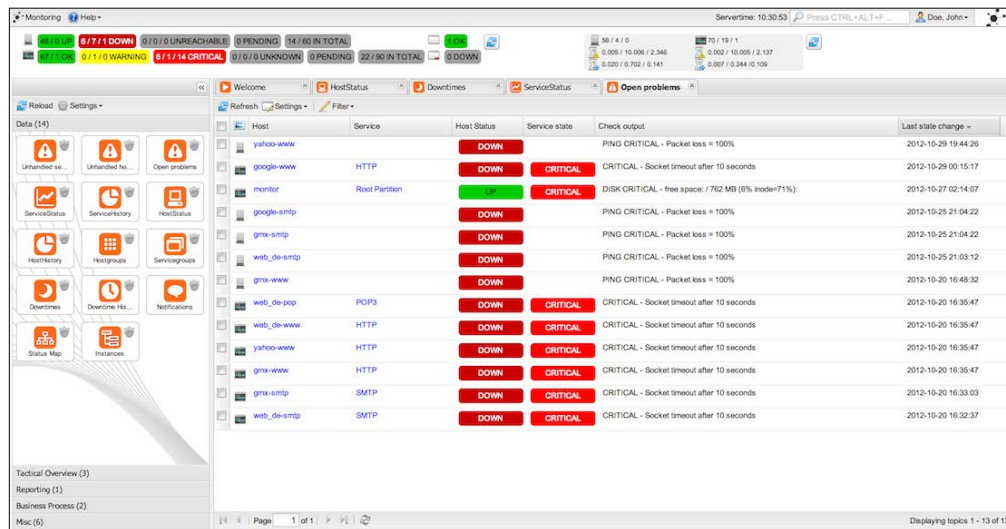
Status view of hosts and services
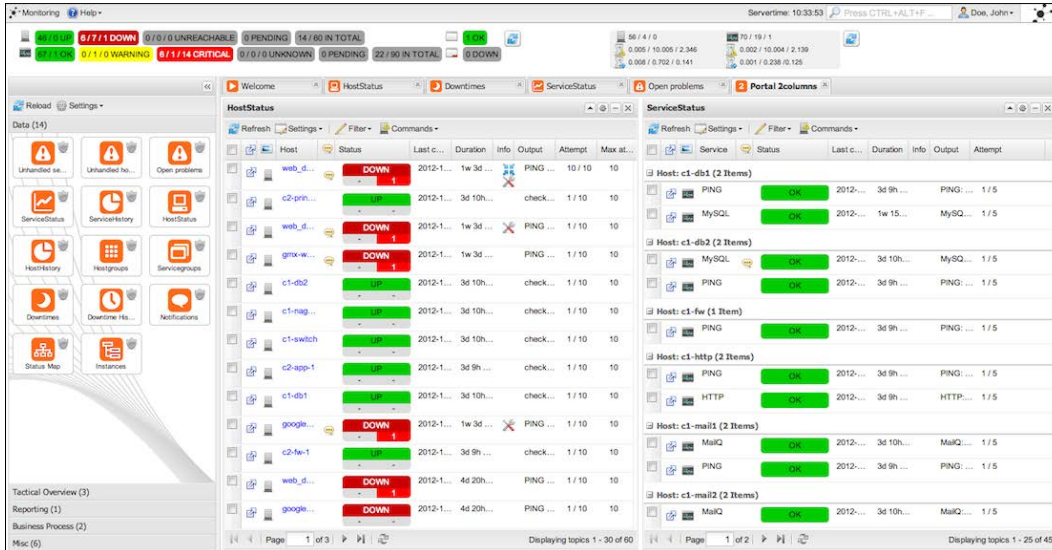
The top part of the interface is similar to the classic web interface. It shows all the host and service statistics by their states. The center view is a little advanced; it has tabs for the user to be able to open multiple views as tabs. The list of available views is on the center left, represented as icons and categorized into several types. Look at the bars at the bottom, each bar represents a category of views.

A tactical overview by hostgroup and servicegroup
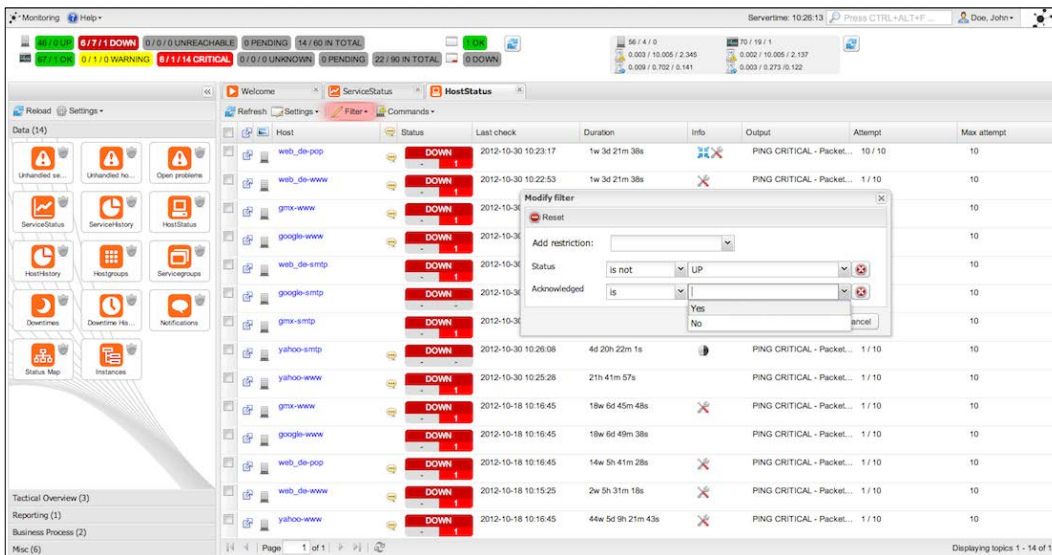
This is similar to the tactical overview view of the classic web interface. It shows the number of hosts and services in different states grouped by hostgroups and servicegroups.

Host and service statuses in a vertically split tab

This is a very convenient view that is split vertically, with host statuses on left and service statuses on right.

Filter entries by various parameters

The status view also has the ability to filter the entries being shown by certain parameters in order to be able to see the most relevant data.

External commands can be executed for multiple services

External command actions are available at the main page itself, which can be run for multiple service checks at once, without having to open up separate page for each service check.

Status map of the network being monitored

This is similar to the status map in the classic web interface. This shows much the same information and some additional statistics on side.

As one can see, all these views can be kept on the interface as tabs, and you can navigate around them without having to go away from the page itself. This greatly improves the usability.

Thruk

Thruk is a web interface with a mix of AJAX and the classic interface. It also uses CGI scripts and has similar views to what we saw earlier. One advantage with Thruk, as mentioned earlier, is that it can connect to multiple Icinga cores.

Suppose we have two Icinga monitoring servers that monitor different sets of networks for some reason. In this case, we will have two Icinga Classic interfaces that we would need to keep checking. Instead, Thruk can connect to both these servers and show all of the information in one interface. This happens via the MK Livestatus broker module available for Nagios, which can be used with Icinga without any change.

Installation and configuration

We need to install `livestatus` on all the monitoring servers. The package installs a `livestatus.o` object file that needs to be loaded into Icinga. This is achieved with the following option in `icinga.cfg`:

```
event_broker_options=-1
broker_module=/usr/local/lib/mk-livestatus/livestatus.o /var/lib/
icinga/rw/live
```

Such a configuration tells Icinga to load the `livestatus` broker module. This module provides a Unix socket file at the specified location (`/var/lib/icinga/rw/live` here). This socket file can be bound to a TCP socket using `xinetd` and `unixcat` and then served over the network. For example, the `xinetd` configuration for this is as follows:

```
service livestatus
{
  type          = UNLISTED
  port          = 6557
  socket_type   = stream
  protocol      = tcp
```

```
    wait          = no
    cps           = 100 3
    instances     = 500
    per_source    = 250
    flags         = NODELAY
    user          = icinga
    server        = /usr/bin/unixcat
    server_args   = /var/lib/icinga/rw/live
    disable       = no
}
```

Now, this Unix socket is now accessible at TCP port 6557.

> **Warning**
>
> The `xinetd` port 6557 mentioned in the preceding code does not require any kind of authentication to access the data, so make sure you have proper firewall rules in place to restrict access to this port.

A remote Thruk installation can now connect to this TCP port and read the Icinga monitoring information it needs to show on the web interface. After installing the `thruk` package, we need to configure the same `htpasswd` file for authentication. The easiest way is to copy Icinga's original `htpasswd` file (or make a symlink to it) in place of Thruk's configured path for the same file that is obtained from Thruk's apache configuration `/etc/httpd/conf.d/thruk.conf`. Another way is to change the path to the `htpasswd` file in this configuration.

Now, we need to configure the backend (peer, in Thruk's terminology) that we want to connect to in Thruk. Add the following lines in `/etc/thruk/thruk_local.conf` depending on your Icinga server backends:

```
<Component Thruk::Backend>
  <peer>
      name    = Icinga 1
      type    = livestatus
      <options>
          peer   = 172.16.0.2:6557
      </options>
  </peer>
  <peer>
      name    = Icinga 2
      type    = livestatus
```
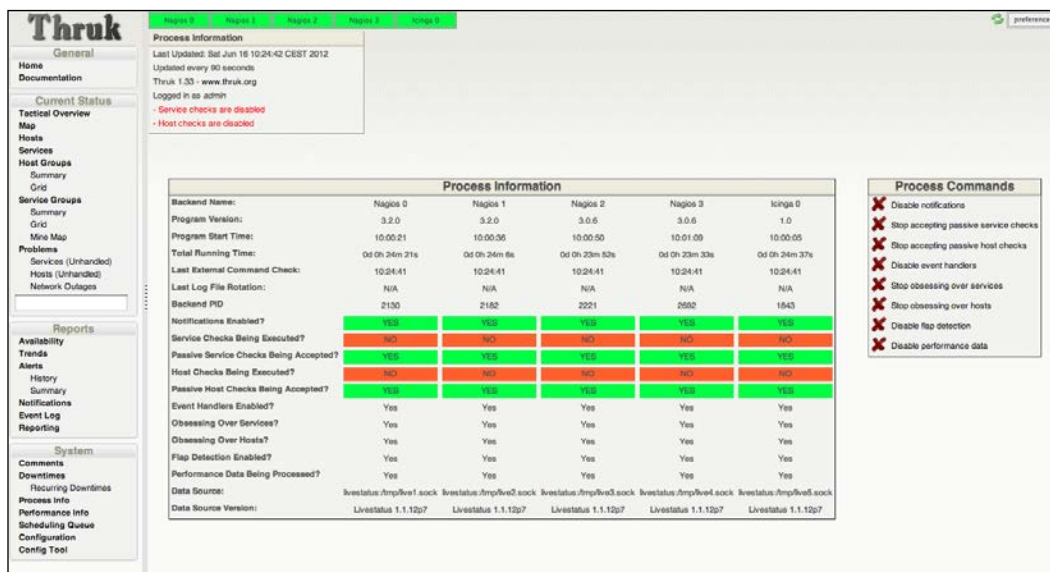
```
        <options>
            peer   = 172.16.0.3:6557
        </options>
    </peer>
</Component>
```

Thruk will connect to the two configured peers at the specified address and port.

Reloading the web server will load Thruk. Open the address where Thruk is installed in a web browser; Thruk is available under the /thruk HTTP path. We should get a screen similar to the following:

Thruk

As we can see, Thruk has a similar navigation bar with similar views. The row with green cells at the top lists the configured peers. This particular example screenshot has five different peers.

Summary

We looked into some of the available web interfaces for Icinga and how each is useful for different purposes. The web interfaces provide easy access to the status overview of the entire network and server infrastructure, and also manages their monitoring to some extent.

With this chapter, we end our overview and insight into Icinga. Icinga evidently is a very flexible monitoring system in terms of configuration, plugins, web interface, and so on. This flexibility can be leveraged to a great extent to suit a variety of simple to complex monitoring requirements. Icinga is a rapidly developing project with some very cool features coming up in next releases. Make sure to keep an eye on the release announcements at their official website (`http://icinga.org`).

Index

Thank you for buying
Icinga Network Monitoring

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.
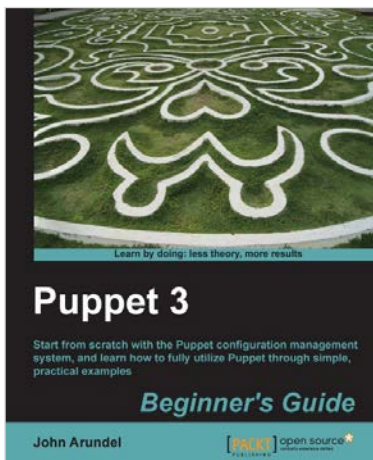
Puppet 3 Beginner's Guide

ISBN: 978-1-78216-124-0   Paperback: 204 pages

Start from scratch with the Puppet configuration management system, and learn how to fully utilize Puppet through simple, practical examples

1. Shows you step-by-step how to install Puppet and start managing your systems with simple examples

2. Every aspect of Puppet is explained in detail so that you really understand what you're doing

3. Gets you up and running immediately, from installation to using Puppet for practical tasks in a matter of minutes
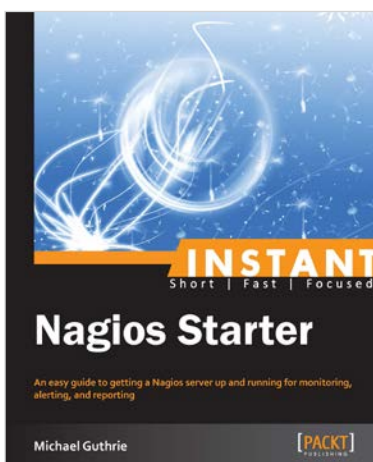
Instant Nagios Starter

ISBN: 978-1-78216-250-6   Paperback: 46 pages

An easy guide to getting a Nagios server up and running for monitoring, alerting, and reporting

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results

2. Install Nagios with minimal fuss on any Unix and Linux platform

3. Harness the flexibility of Nagios for intelligent monitoring

Please check **www.PacktPub.com** for information on our titles
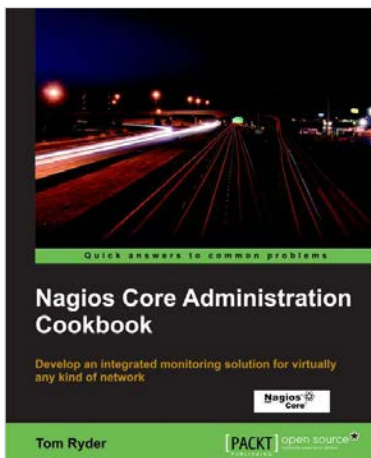
Nagios Core Administration Cookbook

ISBN: 978-1-84951-556-6   Paperback: 360 pages

Develop an integrated monitoring solution for virtully any kind of network

1. Monitor almost anything in a network

2. Control notifications in your network by configuring Nagios Core

3. Get a handle on best practices and time-saving configuration methods for a leaner configuration

BackTrack: Testing Wireless Network Security

ISBN: 978-1-78216-406-7   Paperback: 108 pages

Secure your wireless networks against attacks, hacks, and intruders with this step-by-step guide

1. Make your wireless networks bulletproof

2. Easily secure your network from intruders

3. See how the hackers do it and learn how to defend yourself

Please check **www.PacktPub.com** for information on our titles