



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Mastering Zabbix

Monitor your large IT environment efficiently with Zabbix

Andrea Dalle Vacche
Stefano Kewan Lee

[PACKT] open source*
PUBLISHING community experience distilled

Mastering Zabbix

Monitor your large IT environment efficiently with Zabbix

Andrea Dalle Vacche

Stefano Kewan Lee



Mastering Zabbix

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: December 2013

Production Reference: 1131213

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78328-349-1

www.packtpub.com

Cover Image by John Michael Harkness (jtothem@gmail.com)

Credits

Authors

Andrea Dalle Vacche
Stefano Kewan Lee

Reviewers

Jan Garaj
Nitish Kumar
K. M. Peterson
Gergely Polonkai
Matthew J. Smith

Acquisition Editors

Mary Jasmine Nadar
Nikhil Karkal

Lead Technical Editor

Azharuddin Sheikh

Technical Editors

Sharvari H. Baet
Gauri Dasgupta
Adrian Raposo
Manan Badani
Mrunmayee Patil

Copy Editors

Sarang Chari
Alfida Paiva
Adithi Shetty
Laxmi Subramanian

Project Coordinator

Shiksha Chaturvedi

Proofreader

Mario Cecere

Indexer

Mariammal Chettiyar

Graphics

Yuvraj Mannari
Abhinash Sahu

Production Coordinator

Kyle Albuquerque

Cover Work

Kyle Albuquerque

About the Authors

Andrea Dalle Vacche is a highly skilled IT Professional with over 12 years of industry experience. He graduated from Univerista' degli Studi di Ferrara with an Information Technology certification. This laid the technology foundation, which Andrea has built on ever since. He has acquired various other industry respected accreditations, which include Cisco, Oracle, RHCE, ITIL, and of course Zabbix. Throughout his career he has worked on many large-scale environments, often in roles which have been very complex on a consultant basis. This has further enhanced his growing skill set, adding to his practical knowledge base and concreting his appetite for theoretical technical study. His love for Zabbix came from his time spent in the Oracle world as a Database Administrator/Developer. His time was spent mainly reducing "ownership costs" with specialization in monitoring and automation. This is where he came across Zabbix and the flexibility, both technically and administratively, it offered. Using this as a launch pad, it inspired Andrea to develop Orabbix, the first open source software to monitor Oracle completely integrated with Zabbix.

Andrea has published a number of articles on Zabbix-related software such as DBforBIX. His projects are publicly available on his website <http://www.smartmarmot.com>. Currently, Andrea is working for a leading global investment bank in a very diverse and challenging environment. His involvement is vast and deals with many aspects of the Unix/Linux platforms as well as paying due diligence to many different kinds of third-party software, which are strategically aligned to the bank's technical roadmap.

First, I would like to thank my wife Anna for her support and encouragement during the writing of this book. I highly appreciate her help and advice. Many thanks to Fifi for her relaxing company and fluffy stress relief. I am grateful to my ex-boss Giovanni for his patience when I used to fill his mailbox with odd Zabbix test messages. It was nice having been cheered up by my friends and colleagues: Bav with his precious suggestions and Antonio always ready to encourage me. Special thanks to the Packt Publishing team: Abhijit, Nikhil, Sanhita, Mary, and Reshma. Their advice, effort, and suggestions have been really valuable. The whole team has been very professional and helpful.

Stefano Kewan Lee is an IT Consultant with 10 years of experience in system integration, security, and administration. He is a certified Zabbix specialist in Large Environments, holds a Linux administration certification from the LPI, and a GIAC GCFW certification from SANS Institute. When he's not busy breaking websites, he lives in the countryside with two cats and two dogs and practices martial arts.

I would like to thank all my family and friends for their help and support, my co-author Andrea, and most of all, my partner Roberta for putting up with me on a daily basis.

About the Reviewers

Jan Garaj is a DevOps engineer, who pulls his hair out, when he sees inefficient and expensive server/software solutions. It's because he also wrote some dumb code with terrible exponential $O(n^2)$ memory and computational complexity during software engineering study. However, he learned from a few exceptional effective programmers that good algorithms don't need a ton of memory and four core processors. His first touch with Zabbix was in 2009, when he had to choose a monitoring system for his current employer. He chose Zabbix 1.6. Then, he also came across Zabbix in another company with high website traffic environment or on mission critical servers in the automotive sector. He worked also for Hewlett-Packard, so he knows some proprietary HP monitoring tools. But he still prefers Zabbix for open source stack/web monitoring. His Zabbix love lasts to this day, so he is an active Zabbix frontend language maintainer and he may also be a code contributor one day.

Nitish Kumar is a Wintel Lead at HT Media Ltd. and an independent tech blogger about various technologies. He has been working on several Microsoft technologies and open source solutions (including but not limited to Spiceworks, ManageEngine Products, Zabbix, MS Active Directory, MS Exchange Servers, and so on) for the past eight years, of which the last couple of years have been spent on bringing cost-effective solutions to corporates to simplify their complex requirements and to improve time management for their staff. He is a technology enthusiast and has been participating at various corporate events and public webinars. Mobile technologies have been of special interest to him and he has often written about various gadgets and respective tech. Nitish holds an MS degree in Software from JK Institute of Applied Physics and Technology and his areas of interest include Microsoft technologies, open source software, and mobile gadgets.

He occasionally blogs at <http://nitishkumar.net> and can be reached over e-mail at nitish@nitishkumar.net.

Huge thanks to my wife Pooja and colleagues for being there, and to my kiddo for enduring through the process of writing the book. Thanks to the Packt Publishing team for their persistence and patience—it surely was hard to work with a chaotic person like me.

K. M. Peterson has worked across traditional boundaries in Information Technology, with a passion for listening and teaching, and a focus on helping those who use technology in an organization work more productively and efficiently. His experience spans over 20 years in all facets of Information Technology, across academic, scientific, and commercial environments. He is a technical manager, with hands-on expertise in infrastructure: computer servers, network technologies, data management, and the disciplines of creating facilities that power reliable and scalable applications. He has built data centers and managed teams of systems administrators, database architects, network engineers, and end user support specialists. Mr. Peterson has mastered technologies from operating systems (Windows, Linux, and Mac) to security, Storage Area Networks and virtualization to network management and routing.

He is an independent consultant. Previously, he was Senior Director of Information Technology at Basis Technology Corp., where he architected new applications, enhanced data storage, systems automation, and network services and security. Prior to his position at Basis, Mr. Peterson worked for John Snow, Inc., a global public health consulting firm. Earlier, he managed Information Technology operations for the Whitehead Institute/MIT Center for Genome Research, where he architected the production systems and managed teams that supported the largest contribution of data to the Human Genome Project, crucial to the overall success of one of the largest scientific efforts yet completed.

Gergely Polonkai is a systems engineer for SRV Monitoring, a network monitoring company, and a freelancer software developer. Educated as an electrician, he got in closer touch with computers at the age of 11 while messing with BASIC and Pascal code. In secondary school, he got two Monkey Linux boxes to manage, and his interest and commitment for Open Source Software remains since. Gergely runs a small blog of his own (<http://gergely.polonkai.eu/blog>), which has articles on development and Linux administration.

Matthew J. Smith has been in Enterprise IT since 1996, involved in a myriad of technology areas including systems administration, monitoring, and orchestration, virtualization and cloud, software development, and his favorite area, identity and access management. Through almost two decades of experience at multiple levels of the IT organization, leading teams and committees of all sizes through many tactical and strategic initiatives, Matt has learned the value of bringing open source solutions to enterprise-scale problems.

Most recently, Matt has taken the opportunity to join Red Hat, Inc., as a Solution Architect, bringing his passion for and expertise in Enterprise Open Source to those looking to leverage Red Hat technologies to solve real business problems. Interacting daily with enterprises in a great variety of global industries, including financial services, insurance, healthcare, pharmaceutical, automotive services, manufacturing, and hospitality, Matt experiences the viability and power of Open Source solutions solving the most complex problems. Matt has come to truly appreciate the values of Open Source, is a contributing member of multiple Open Source communities, and is a vocal advocate for Open Source approaches to solving problems.

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	1
Chapter 1: Deploying Zabbix	9
Defining the environment size	10
Zabbix architectures	11
Zabbix installation	14
Prerequisites	16
Setting up the server	17
Setting up the agent	18
Installing and creating the package	19
Configuring the server	20
Installing the database	21
Starting up with Daemon	24
Some considerations about the database	30
Sizing of the database	32
Some considerations about housekeeping	35
The web interface	41
The web wizard – frontend configuration	42
Capacity planning with Zabbix	48
The observer effect	48
What to monitor	48
Define a baseline	50
Load testing	51
Forecasting the trends	53
Summary	54
Chapter 2: Distributed Monitoring	55
Zabbix proxies	56
Deploying a Zabbix proxy	57
Understanding the flow of monitoring data with proxies	63
Zabbix nodes	66
Understanding the flow of data with nodes	67
Deploying a node	67

Proxies versus nodes	72
Disadvantages of nodes	72
Choosing between proxies and nodes	73
Security considerations	73
No network configuration	74
Network isolation	76
Simple tunnels	76
Secure Shell	76
Stunnel	78
A full-blown VPN	79
Summary	80
Chapter 3: High Availability and Failover	81
Understanding high availability	82
Understanding the levels of IT service	83
Some consideration about high availability	84
Automating the switchover/the failover with a resource manager	86
Replicating the filesystem with DRBD	86
Implementing high availability on a web server	87
Configuring HTTPD HA	88
Understanding Pacemaker and STONITH	90
Pacemaker – is Quorum really needed?	91
Pacemaker – the stickiness concepts	92
Pacemaker – the Apache/HTTPD configuration	92
Configuring the Zabbix Server for high availability	96
Database high availability	98
Clustering of PostgreSQL	100
Mirrored logical volume with LVM and DRBD	100
Prerequisite tasks to start with DRBD on LVM	102
Creating a DRBD device on top of the LVM partition	103
Enabling resource in DRBD	104
Defining a primary device in DRBD	105
Creating a filesystem on a DRBD device	105
Pacemaker cluster – integrating DRBD	107
Enabling the DRBD configuration	107
Pacemaker – the LVM configuration	108
Pacemaker – configuring PostgreSQL	109
Pacemaker – the network configuration	109
Pacemaker – the final configuration	110
Cluster configuration – the final test	110
DRBD performance and optimizations	112
Efficient synchronization by DRBD	112
Enabling online verification through DRBD	114
DRBD – some networking consideration	115
Summary	118

Chapter 4: Collecting Data	119
Gathering items as raw data	120
Understanding the data flow for Zabbix items	122
Understanding Zabbix trapper items	125
The data flow overview	125
Database monitoring with Zabbix	126
Delving into ODBC	127
Installing database drivers	128
MySQL ODBC drivers	128
PostgreSQL ODBC drivers	130
Oracle ODBC drivers	131
UnixODBC configuration files	133
Compiling Zabbix with ODBC	134
Database monitor items	135
Some considerations about the ODBC SQL query	136
Zabbix JMX monitoring	137
Considering some JMX security aspects	138
Installing a Zabbix Java gateway	139
Configuring Zabbix JMX	141
JMX keys in detail	143
Issues and considerations about JMX	144
Zabbix SNMP monitoring	145
SNMP queries	148
SNMP traps	151
The snmptrapd process	152
The Perl trap handler	153
Web pages monitoring	158
Authenticating on web pages	160
Logging out	162
Aggregated and calculated items	164
Aggregated items	164
Calculated items	167
Summary	168
Chapter 5: Visualizing Data	169
Graphs	171
Analyzing simple graphs	171
Analyzing custom graphs	174
Reviewing all combinations of graph properties	179
Visualizing the data through maps	182
Creating your first Zabbix Map	186
Some important considerations about macros and URLs	188
Finally inside the map	191
Selecting elements	194

Playing with macros inside maps	195
Visualizing through screens	197
Creating a screen	198
Dynamic elements	199
Visualizing the data through a slide show	201
Controlling centre slides and the big display challenge	202
Some consideration about slides on a big display	203
Automated slide show	204
IT services	205
Configuring an IT service	206
Summary	209
Chapter 6: Managing Alerts	211
Understanding trigger expressions	212
Selecting items and functions	213
Choosing between seconds or number of measurements	214
Date and time functions	215
Trigger severity	216
Choosing between absolute values and percentages	216
Understanding operations as correlations	217
Managing the trigger dependencies	221
Taking action	222
Defining an action	223
The {EVENT.DATE} and {EVENT.TIME} macros	224
The {INVENTORY.SERIALNO.A} and friends macros	224
The {NODE.ID} and {NODE.NAME} macros	224
Defining the action conditions	225
Choosing the action operations	226
Steps and escalations	227
Messages and media	228
Remote commands	230
Summary	231
Chapter 7: Managing Templates	233
Creating templates	234
Adding entities to a template	234
Using macros	236
User-defined macros	241
Linking templates to hosts	242
Nesting templates	242
Combining templates	244
Discovering hosts	244
Low-level discovery	248
Summary	252

Chapter 8: Handling External Scripts	253
External checks	254
The script's placement	255
Going deep into external checks	255
Inside the script	258
General script writing rules	260
Some consideration about external checks	260
The UserParameter	261
The flexible UserParameter	261
Some considerations about UserParameters	263
Sending data using zabbix_sender	264
The new script	265
Writing a wrapper script for check_ora_sendtrap	266
Pros and cons of a dedicated script server	267
Working with Zabbix protocols	268
The Zabbix get protocol	269
The Zabbix sender protocol	269
An interesting undocumented feature	271
Using clock properties in JSON items	272
The Zabbix Agent protocol	273
Some more possible responses	275
Communicating with Zabbix	276
Implementing Zabbix sender in Java	276
Implementing Zabbix sender in Python	278
Some considerations about agent development	280
Summary	281
Chapter 9: Extending Zabbix	283
Exploring the Zabbix API	284
Authenticating through the API	286
Using the PyZabbix library	288
Synchronizing templates across different nodes	291
Mass operations	294
Redistributing hosts to proxies	294
Adding or updating users	295
Exporting data	298
Extracting tabular data	299
Creating graphs from data	302
The Graphviz suite of programs	302
Creating a trigger dependency graph	303
Summary	306

Chapter 10: Integrating Zabbix	307
An overview of Request Tracker	308
Setting up RT to better integrate with Zabbix	309
Creating a custom queue for Zabbix	310
Customizing tickets – the links section	311
Customizing tickets – ticket priority	312
Customizing tickets – the custom fields	313
Connecting to the Request Tracker API	315
Setting up Zabbix to integrate with Request Tracker	318
Creating RT tickets from the Zabbix events	322
Summary	328
Index	329

Preface

Ever since its first public release in 2001, Zabbix has distinguished itself as a very powerful and effective monitoring solution. As an open source product, it's easy to obtain and deploy, and its unique approach to metrics and alarms has helped to set it apart from its competitors, both open and commercial. It's a powerful, compact package with very low requirements in terms of hardware and supporting software for a basic yet effective installation. If you add a relative ease of use, it's clear that it can be a very good contender for small environments with a tight budget. But it's when it comes to managing a huge number of monitored objects, with a complex configuration and dependencies, where Zabbix's scalability and inherently distributed architecture really shines. More than anything, Zabbix can be an ideal solution in large, complex, distributed environments, where being able to manage efficiently and extract meaningful information from monitored objects and events is just as important, if not more important, than the usual considerations about costs, accessibility, and the ease of use.

The purpose of this book is to help make the most of your Zabbix installation to leverage all of its power to monitor effectively any large and complex environment.

What this book covers

Chapter 1, Deploying Zabbix, will focus on choosing the optimal hardware and software configuration for the Zabbix server and database in relation to the current IT infrastructure, monitoring goals, and possible evolution. This chapter also includes a section that covers an interesting database-sizing digression, useful to calculate the final database size using a standard environment as the baseline. Correct environment sizing and a brief discussion about metrics and measurements that can also be used for capacity planning will be covered here. The chapter will contain practical examples and calculations framed in a theoretical approach to give the reader the skills required to adapt the information to real-world deployments.

Chapter 2, Distributed Monitoring, will explore the various Zabbix components, both on the server and agent side. In addition to the deployment and configuration of agents, proxies and nodes, maintenance, changed management, and security will all be taken into account. This section will cover all the possible Zabbix architectural implementations adding the pros and cons considerations.

Chapter 3, High Availability and Failover, will cover the subjects of high-availability and failover. For each of the three main Zabbix tiers, the reader will learn to choose among different HA options. The discussion will build on the information provided in the previous two chapters, in order to end the first part of the book with a few complete deployment scenarios that will include high-availability server and databases hierarchically organized in tiered, distributed architectures geared at monitoring thousands of objects scattered in different geographical locations.

Chapter 4, Collecting Data, will move beyond simple agent items and SNMP queries to tackle a few complex data sources. The chapter will explore some powerful Zabbix built-ins, how to use them, and how to choose the best metrics to ensure thorough monitoring without overloading the system. There will also be special considerations about aggregated values and their use to monitor complex environments with clusters or the more complex grid architectures.

Chapter 5, Visualizing Data, will focus on getting the most out of the data visualization features of Zabbix. This one is a quite useful section especially if you need to explain or chase some hardware expansion/improvement to the business unit. You will learn how to leverage live monitoring data to make dynamic maps and how to organize a collection of graphs for big-screen visualization in control centers and implement a general qualitative view. This chapter will cover completely the data center *quality view* slide show, which is really useful to highlight problems and warn the first-level support in a proactive approach. The chapter will also explore some best practices concerning the IT services and SLA reporting features of Zabbix.

Chapter 6, Managing Alerts, will give examples of complex triggers and trigger conditions, as well as some advice on choosing the right amount of trigger and alerting actions. The purpose is to help you walk the fine line between being blind to possible problems and being overwhelmed by false positives. You will also learn how to use actions to automatically fix simple problems, raising actions without the need of human intervention to correlate different triggers and events, and how to tie escalations to your operations management workflow. This section will make you aware of what can be automated, reducing your administrative workload and optimizing the administration process in a proactive way.

Chapter 7, Managing Templates, will offer some guidelines for effective template management: building complex templates schemes out of simple components, understanding and managing the effects of template modification and maintenance of existing monitored objects, and assigning templates to discovered hosts. This will conclude the second part of the book, dedicated to the different Zabbix monitoring and data management options. The third and final part will discuss Zabbix's interaction with external products and all its powerful extensibility features.

Chapter 8, Handling External Scripts, will help you learn how to write scripts to monitor objects not covered by the core Zabbix features. The relative advantages and disadvantages of keeping the scripts on the server side or agent side, how to launch or schedule them, and a detailed analysis of the Zabbix agent protocol will also be covered. This section will make you aware of all the possible side effects, delay, and load caused by script; you will so be able to implement all the needed external checks, well aware of all that is connected with them and the relative *observer effect*. The chapter will include different implementations on working Bash, Java, and Python so that you can easily write your own scripts to extend and enhance Zabbix's monitoring possibilities.

Chapter 9, Extending Zabbix, will delve into the Zabbix API and how to use it to build specialized frontends and complex extensions or to harvest monitoring data for further elaboration and reporting. It will include some simple example implementations, written in Python, that will illustrate how to export and further manipulate data; how to perform massive and complex operations on monitored objects; and finally, how to automate different management aspects like user creation and configuration, trigger activation, and the like.

Chapter 10, Integrating Zabbix, will wrap things up discussing how to make other systems know about Zabbix, and the other way around. This is key to the successful management of any large or complex environment. You will learn how to use built-in Zabbix features, API calls, or direct database queries to communicate with different upstream and downstream systems and applications. To further illustrate the integration possibilities, there will be a complete and concrete example of interaction with the Request Tracker trouble-ticket system.

Who this book is for

As the book title is *Mastering Zabbix*, you won't find any detailed, step-by-step tutorials (well, except the installation that will be covered from scratch, but with some useful tips) on the basic usage of Zabbix. Although you may find lots of detailed information about installing the server, or configuring items, triggers, and screens, you are expected to have at least basic working knowledge of how it all works, so that you can focus on a more advanced approach for the same subjects. That said, it is possible to profit from the contents of this book even if you have no previous Zabbix exposure, but in that case, you are strongly encouraged to refer to the official Zabbix documentation that you can find at <https://www.zabbix.com/documentation/2.0/manual> to fill in any possible gaps in your knowledge.

What you need for this book

Before going deep into the Zabbix setup, it is important to know that the proposed setup covered here are tested on a large-production environment (more than 1800 hosts monitored, more than 89500 monitored items, and more than 30000 triggers) and they can be considered valid for most of large and very large environments. The high-availability solution proposed in this book has been widely tested, not purely as a disaster recovery exercise but during a real disaster (network cables accidentally sheared by an excavating machine).

In this book, it is important to understand that most of the choices done has been on a practical basis and not driven by passion. One of the main choices made is using PostgreSQL as the official Zabbix RDBMS. We came across PostgreSQL as RDBMS mostly for the mature and production ready, features offered:

- Hot backup is available by design
- Atomicity, consistency, isolation, and durability: in one word it is fully ACID compliance
- Many different native standby configurations (hot standby, synchronous replication, and so on)
- Efficient partitioning

Zabbix's database is a critical component, especially if you need to keep historical data available and guarantee constant performances day by day while the database is growing.

We have made some assumptions in this book: the packaging system used in our examples is yum and then the distribution is obviously a Red Hat Enterprise Linux. Anyway, excluding some details such as package names and packet manager, the whole book is valid for all the Linux distributions. Furthermore, the proposed architectures and their implementation are not directly tied to a particular distribution. We did not use any Red Hat-specific clustering system or a choice that you cannot reproduce on your favorite Linux distribution.

On reading this book, you will find different open source software, but between all of them it would be better if you are familiar in particular with the following:

- **Apache:** <http://www.apache.org/>
- **Pacemaker:** <http://clusterlabs.org/>
- **PostgreSQL:** <http://www.postgresql.org/>
- **DRBD:** <http://www.drbd.org>

This book also focuses on system administrators that have some programming skills. We propose different workings for the implemented code snippet. With the proposed examples all well documented, you should be able to implement your own plugin or external software fully integrated with Zabbix. The code snippets proposed are on two different and widely diffused languages: Java and Python. These cover most of the current programmers' preferences and show, once you know how to implement the Zabbix protocol, how simple it is to switch between them.

Zabbix is more than monitoring software; it is an open source monitoring solution that can be explained as you want, and this book will make you aware of all the pros and cons for the possible solutions.

So now it is time to go deep into the Zabbix's land!

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 use yum (from root) and install the following package."

A block of code is set as follows:

```
# ./configure --enable-server --with-postgresql --with-libcurl
--with-jabber --with-net-snmp --enable-ipv6 --with-openipmi --with-
ssh2 --with-ldap --enable-agent
```


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


```
# groupadd zabbixzabbix
# useradd -m -s /bin/bash -g zabbix zabbix
# useradd -m -s /bin/bash -g zabbix zabbixsvr
```

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

```
# yum list postgres*
```

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: "The first screen that you will meet is a welcome page; there is nothing to do there other than to click on **Next**".

[ 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.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

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

Deploying Zabbix

Most probably, if you are reading this book, you have already used and installed Zabbix. Most likely on a small/medium environment, but now things have changed and your environment today is a large one with new challenges coming in regularly. Nowadays environments are rapidly growing or changing and it is a difficult task to be ready to support and provide a reliable monitoring solution.

Normally, an initial startup of a system, also a monitoring system, is done by following some tutorial or how-to and this is a common error. This kind of approach is valid for smaller environments, when the downtime is not critical, when there are no disaster recovery sites to handle, or, in few words, when things are easy.

Most likely these setups are not done looking forward to the possible new quantity of new items, triggers, and events that the server should elaborate. If you have already installed Zabbix and you need to plan and expand your monitoring solution, or if instead you need to plan and design the new monitoring infrastructure, this chapter will help you.

This chapter will also help you perform the difficult task of setting up/upgrading Zabbix in a large and very large environment. This chapter will cover every aspect of this task, starting with the definition of a large environment up to using Zabbix as a capacity planning resource. The chapter will introduce all the possible Zabbix solutions, doing a practical example with an installation ready to handle a large environment, going ahead with some possible improvements.

At the end of this chapter, you will understand how Zabbix works, which tables should be kept under special surveillance and you will know how to improve the housekeeping on a large environment that, with some years of trends to handle, is a really heavy task.

This chapter will cover the following points:

- Knowing when you are in front of a large environment and defining when an environment can be considered a large environment
- Setting up/upgrading Zabbix on a large and a very large environment
- Installing Zabbix on a three tier setup and having a readymade solution to handle a large environment
- Database sizing and finally knowing the total amount of space consumed by our data acquired
- Knowing the database heavy tables and tasks
- Improving housekeeping to reduce the RDBMS load and improving the efficiency of the whole system
- Learning some fundamental concepts about capacity planning, bearing in mind that Zabbix is and can be a capacity-planning tool

Defining the environment size

Since this book is focused on a large environment, we need to define or at least provide some basic fixed points to identify a large environment. There are various things to consider in this definition; basically, we can identify an environment as large when:

- There are more different physical locations
- The number of monitored devices is high (hundreds or thousands)
- The number of checks and items retrieved per second (more than 500)
- There is a lot of data to handle (database is larger than 100 GB)
- The availability and performance are both critical

All of the preceding points define a large environment; in this kind of environment, the installation and maintenance of Zabbix infrastructure play a critical role.

The installation of course is a task well-defined on a timely basis and probably one of the most critical tasks; it is really important to go live with a strong and reliable monitoring infrastructure. Also, once we go live with the monitoring in place, it will not be so easy to move/migrate pieces without any data loss. There are some other things to consider: we will have a lot of tasks, most of them are daily tasks but in a large environment they require particular attention.

In a small environment with a small database, a backup will keep you busy for few minutes, but if the database is quite large, this task will consume a considerable quantity of time.

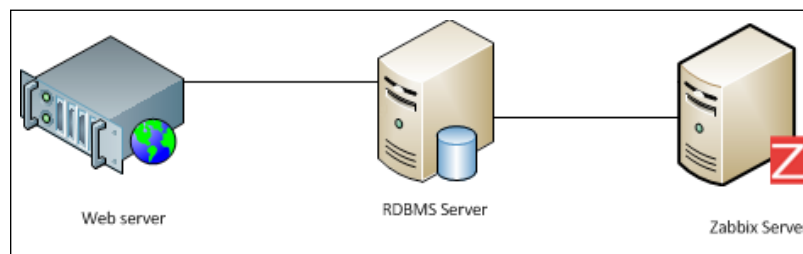
The restore and a relative restore plan should be considered and tested periodically to be aware of the time needed to complete this task in case of a disaster or critical hardware failure.

Between the maintenance tasks, we need to consider testing and putting on production the upgrades with a minimal impact and all the daily tasks and daily checks.

Zabbix architectures

Zabbix can be defined as a distributed monitoring system with a centralized web interface (on which we can manage almost everything). Among its main features, we will highlight the following ones:

- It has a centralized web interface
- The server can be run on most Unix-like operating systems
- We can use native agents for most of the Unix, Unix-like, and Microsoft Windows operation systems
- It has an extremely flexible configuration
- It is easy to integrate with other systems
- It can be monitored via SNMP (v1, v2, and v3), IPMI, JMX, ODBC, and SSH
- It gives us a possibility to create custom items, graphs, and interpolate data
- It is easy to customize

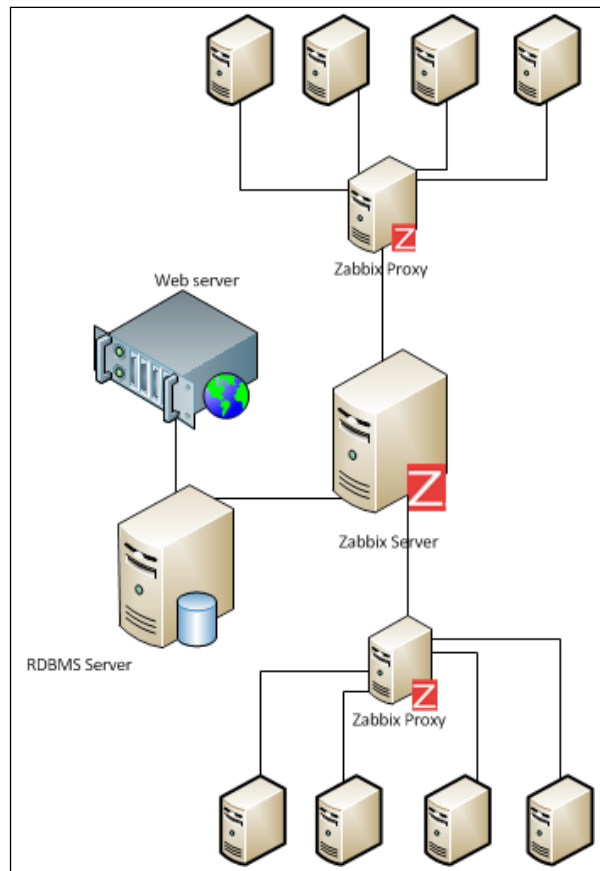


The Zabbix architecture for a large environment is composed of three different servers/components (that should be configured on HA as well). These three components are as follows:

- A web server
- A Zabbix server
- An RDBMS server

The whole Zabbix infrastructure in large environments allows us to have two other actors that play a fundamental role. These actors are the Zabbix agents and the Zabbix proxies.

An example is represented in the following figure:



On this infrastructure, we have a centralized Zabbix server that is connected to different proxies, usually one for each server farm or a subnetwork.

The **Zabbix server** will acquire data from **Zabbix proxies** and the proxies will acquire data from all the **Zabbix agents** connected to it, all the data is stored on a dedicated RDBMS, and the frontend is exposed with a web interface to the users. Looking at the technologies used, we see that the web interface is written in PHP and the server, proxies, and agents are written in C.



The server, proxies, and agents are written in C to give the best performance and less resource usage possible. All the components are deeply optimized to achieve the best performance.

We can implement different kinds of architecture using proxies. There are several type of architectures and in order of complexity we find:

- The single server installation
- One server and many proxies
- Distributed installation

The single server installation is not suggested on a large environment. It is the basic installation where single servers do the monitoring and it can be considered a good starting point.

Most probably, in our infrastructure, we might already have a Zabbix installation. Zabbix is quite flexible and this permits us to upgrade this installation to the next step: proxy-based monitoring.

Proxy-based monitoring is implemented with one Zabbix server and several proxies, that is one proxy per branch or datacenter. This configuration is easy to maintain and offers an advantage to have a centralized monitoring solution. This kind of configuration is the right balance between large environment monitoring and complexity. From this point, we can (with a lot of effort) expand our installation to a complete and distributed monitoring architecture. The installation composed by one server and many proxies is the one shown in the previous diagram.

The distributed scenarios are the most complex installation that we can implement with Zabbix. This scenario is composed of one Zabbix server per branch (or datacenter). This installation is the most complex to set up and maintain and unfortunately is not centralized but can be combined with proxies.

All the possible Zabbix architecture will be discussed in detail in *Chapter 2, Distributed Monitoring*.

Zabbix installation

The installation that will be covered in this chapter is the one composed by a server for each of the following base components:

- A web frontend
- A Zabbix server
- A Zabbix database

We will start describing this installation because:

- It is a basic installation that is ready to be expanded with proxies and nodes
- Each component is on a dedicated server
- This kind of configuration is the starting point for monitoring large environments
- It is widely used
- Most probably it will be the starting point of your upgrade and expansion of monitoring infrastructure.

Actually, this first setup for a large environment, as explained here, can be useful if you have the task to improve an existing monitoring infrastructure. If your current monitoring solution is not implemented in this way, the first thing is to plan the migration on three different dedicated servers.

Once the environment is set up on three tiers, but is still giving poor performance, you can plan and think which kind of large environment setup will be a perfect fit for your infrastructure.

When you monitor your large environment, there are some points to consider:

- Use a dedicated server to keep the things easy-to-extend
- Easy-to-extend and implement a high availability setup
- Easy-to-extend and implement a fault tolerant architecture

On this three-layer installation, the CPU usage of server component will not be really critical, at least for the Zabbix server. The CPU consumption is directly related to the number of items to store and the refresh rate (number of samples per minute) rather than the memory.

Indeed the Zabbix server will not consume excessive CPU but is a bit more greedy of memory. We can consider that a quad core server and 4 GB of RAM can be used for more than 1000 hosts without any issues.

Basically, there are two ways to install Zabbix:

- Downloading the latest source code and compiling it
- Installing it from packages

There is also another way to have a Zabbix server up and running, that is by downloading the virtual appliance but we don't consider this case as it is better to have full control of our installation and be aware of all the steps.

The installation from packages gives us the following benefits:

- It makes the process of upgrading and updating easier
- Dependencies are automatically sorted

The source code compilation also gives us some benefit:

- We can compile only the required features
- We can statically build the agent and deploy it on different Linux "flavors"
- We can have complete control on the update

It is quite usual to have different versions of Linux, Unix, and Microsoft Windows on a large environment. These kind of scenarios are quite diffused on a heterogeneous infrastructure; and if we use the agent distribution package of Zabbix on each Linux server, we will for sure have different versions of the agent and different locations for the configuration files.

The more standardized we are across the server, the easier it will be to maintain and upgrade the infrastructure. `--enable-static` give us a way to standardize the agent across different Linux versions and releases and this is a strong benefit. The agent if statically compiled can be easily deployed everywhere and for sure we will have the same location (and we can use the same configuration file apart from the node name) for the agent and his configuration file. The deployment will be standardized; however, the only thing that may vary is the start/stop script and how to register it on the right init runlevel.

The same kind of concept can be applied to commercial Unix bearing in mind its compilation by vendors, so the same agent can be deployed on different versions Unix release by the same vendor.

Prerequisites

Before compiling Zabbix, we need to take a look at the prerequisites. The web frontend will need at least the following version:

- Apache (1.3.12 or later)
- PHP (5.1.6 or later)

Instead, the Zabbix server will need:

- An RDBMS – The open source alternatives are PostgreSQL and MySQL
- zlib-devel
- mysql-devel – It is used for supporting MySQL (not needed on our setup)
- postgresql-devel – It is used for supporting PostgreSQL
- glibc-devel
- curl-devel – It is used on web monitoring
- libidn-devel – The curl-devel depends on it
- openssl-devel – The curl-devel depends on it
- net-snmp-devel – It is used on SNMP support
- popt-devel – The net-snmp-devel might depend on it
- rpm-devel – The net-snmp-devel might depend on it
- OpenIPMI-devel – It is used for supporting IPMI
- iksemel-devel – it is used for jabber protocol
- Libssh2-devel
- sqlite3 – It is required if SQLite is used as Zabbix backend database (usually on proxies)

On a Red Hat Enterprise Linux distribution, to install all the dependencies, we can use yum (from root) and install the following package:

```
yum install zlib-devel postgresql-devel glibc-devel curl-devel gcc  
automake postgresql libidn-devel openssl-devel net-snmp-devel rpm-  
devel OpenIPMI-devel iksemel-devel libssh2-devel openldap-devel
```



iksemel-devel is used to send a Jabber message. This is a really useful feature as it enables Zabbix to send chat messages. Furthermore, Jabber is managed as a media type on Zabbix and you can also set your working time, which is a really useful feature to avoid the sending of messages when you are not in office.

Setting up the server

Zabbix needs a user and an unprivileged account to run. Anyway, if the daemon is started from root, it will automatically switch to the Zabbix account if this one is present.

```
# groupadd zabbix
# useradd -m -s /bin/bash -g zabbix zabbix
# useradd -m -s /bin/bash -g zabbix zabbixsvr
```



The server should never run as root because it will expose the server to a security risk.

The preceding lines permit you to enforce the security of your installation. The server and agent should run with two different accounts otherwise the agent can access Zabbix server's configuration. Now, using the Zabbix user account we can extract the tar.gz file:

```
# tar -zxvf zabbix-2.0.6.tar.gz
```

To know all the possible configuration options you can check use the `--help` parameter as follows

```
# ./configure --help
```

To configure the source for our server, we can use the following options:

```
# ./configure --enable-server --with-postgresql --with-libcurl
--with-jabber --with-net-snmp --enable-ipv6 --with-openipmi --with-
ssh2 --with-ldap --enable-agent
```



The `zabbix_get` and `zabbix_send` command lines are generated only if `--enable-agent` is specified during server compilation.

If the configuration is complete without errors, we should see something like this:

```
Enable server:          yes
Server details:
  With database:        PostgreSQL
  WEB Monitoring via:   cURL
  Native Jabber:        yes
  SNMP:                 net-snmp
  IPMI:                 openipmi
  SSH:                  yes
  ODBC:                  no
```

```
Linker flags:          -rdynamic          -L/usr/lib64          -L/usr/
lib64 -L/usr/lib64 -L/usr/lib -L/usr/lib -L/usr/lib
Libraries:             -lm -lrt -lresolv          -lpq -liksemel
-lcurl -lnetsnmp -lcrypto -lnetsnmp -lcrypto -lssh2 -lOpenIPMI
-lOpenIPMIposix -lldap -llber

Enable proxy:          no

Enable agent:          yes
Agent details:
Linker flags:          -rdynamic          -L/usr/lib
Libraries:             -lm -lrt -lresolv          -lcurl -lldap -llber

Enable Java gateway:   no

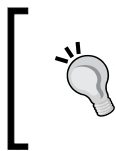
LDAP support:          yes
IPv6 support:          yes
*****
*           Now run 'make install'           *
*                                           *
*           Thank you for using Zabbix!       *
*           <http://www.zabbix.com>          *
*****
```

We will not run `make install` but only the compilation with `# make`. To specify a different location for the Zabbix server, we need to use a `--prefix` on the configure options, for example, `--prefix=/opt/zabbix`. Now, follow the instructions as explained in the *Installing and creating the package* section.

Setting up the agent

To configure the sources to create the agent, we need to run the following command:

```
# ./configure --enable-agent
# make
```



With the `make` command followed by the `--enable-static` option, you can statically link the libraries and the compiled binary will not require any external library; this is very useful to distribute the agent across a different dialect of Linux.

Installing and creating the package

In both the previous sections, the command line ends right before the installation, indeed we didn't run the following command:

```
# make install
```

I advise you to not run the `make install` command but use the `checkinstall` software instead. This software will create the package and install the Zabbix software.

You can download the software from <http://asic-linux.com.mx/~izto/checkinstall/>.

Note that `checkinstall` is only one of the possible alternatives you have to create a distributable system package.



We can also use a prebuild `checkinstall`. The current release is `checkinstall-1.6.2-20.2.i686.rpm` (on Red Hat/CentOS); the package will also need `rpm-build`:

```
yum install rpm-build
```

And we need to create the necessary directories:

```
mkdir -p ~/rpmbuild/{BUILD,RPMS,SOURCES,SPECS,SRPMS}
```

The package made things easy; it is easy to distribute and upgrade the software plus we can create a package for different versions of a package manager: RPM, deb, and tgz.



`checkinstall` can produce a package for Debian, Slackware, and RedHat. This is particularly useful to produce the Zabbix's agent package (statically linked) and to distribute it around our server.

Now we need to convert to root or use the `sudo checkinstall` command followed by its options:

```
# checkinstall --nodoc --install=yes -y
```


If you don't face any issue, you should get the following message:

```
*****
Done. The new package has been saved to

/root/rpmbuild/RPMS/i386/zabbix-2.0.6-1.i386.rpm
You can install it in your system anytime using:

rpm -i zabbix-2.0.6-1.i386.rpm
*****
```

Now Zabbix is installed. The server binaries will be installed into `<prefix>/sbin`, utilities will be on `<prefix>/bin`, and the man pages under the `<prefix>/share` location.

Configuring the server


For the server configuration, we only have one file to check and edit:

```
/usr/local/etc/zabbix_server.conf
```

The configuration files are inside the following directory:

```
/usr/local/etc
```

We need to change the `/usr/local/etc/zabbix_server.conf` file and write the username, relative password, and the database name there.

 This location depends on the `sysconfdir` compile-time installation variable. Don't forget to take appropriate measures to protect the access to the configuration file with the following command:


```
chmod 400/usr/local/etc/zabbix_server.conf
```

The location of the default external scripts will be:

```
/usr/local/share/zabbix/externalscripts
```

It depends on the `datadir` compile-time installation variable. The `alertscripts` directory will be at the following location:

```
/usr/local/share/zabbix/alertscripts
```

 This can be changed during compilation and it depends on the `datadir` installation variable.

Now we need to configure the agent. The configuration file is where we need to write the IP address of our Zabbix server. Once done, it is important to add two new services to the right runlevel to be sure that they will start when the server enters on the right runlevel.

To complete this task we need to install the start/stop scripts on the following:

- `/etc/init.d/zabbix-agent`
- `/etc/init.d/zabbix-proxy`
- `/etc/init.d/zabbix-server`

There are several scripts prebuilt inside the `misc` folder located at the following location:

```
zabbix-2.0.6/misc/init.d
```

This folder contains different startup scripts for different Linux variants, but this tree is not actively maintained and tested, and may not be up-to-date with the most recent versions of Linux distributions so it is better to take care and test it before going live.

Once the start/stop script is added inside the `/etc/init.d` folder, we need to add it to the service list:

```
# chkconfig --add zabbix-server
# chkconfig --add zabbix-agentd
```

Now all that is left is to tell the system on which runlevel it should start them, we are going to use runlevel 3 and 5.

```
# chkconfig --level 35 zabbix-server on
# chkconfig --level 35 zabbix-agentd on
```

Currently, we can't start the server; before starting up our server we need to configure the database.

Installing the database

Once we complete the previous step, we can walk through the database server installation. All those steps will be done on the dedicated database server. The first thing to do is install the PostgreSQL server. This can be easily done with the package offered from the distribution but is recommended to use the latest 9.x stable version.

RedHat is still distributing the 8.x on RHEL6.4. Also, its clones such as CentOS and ScientificLinux are doing the same. PostgreSQL 9.x has many useful features, at the moment, the latest stable ready for the production environment is Version 9.2.

To install the PostgreSQL 9.2, there are some easy steps to follow:

1. Locate the .repo files:
 - **Red Hat:** It is present at `/etc/yum/pluginconf.d/rhnplugin.conf` [main]
 - **CentOS:** It is present at `/etc/yum.repos.d/CentOS-Base.repo`, [base] and [updates]
2. Append the following line on the section(s) identified in the preceding step:
`exclude=postgresql*`
3. Browse to `http://yum.postgresql.org` and find your correct RPM. For example, to install PostgreSQL 9.2 on the CentOS 6 32 bit, go to `http://yum.postgresql.org/9.2/redhat/rhel-6-i386/pgdg-redhat92-9.2-7.noarch.rpm`.
4. Now to list the entire postgresql package, use the following command:
`# yum list postgres*`
5. Once you find our package in the list, install it using the following command:
`# yum install postgresql92 postgresql92-server postgresql92-contrib`
6. Once the packages are installed, we need to initialize the database:
`# service postgresql-9.2 initdb`

Or:

`# /etc/init.d/postgresql-9.2 initdb`
7. Now we need to change a few things in the configuration file `/var/lib/pgsql/9.2/data/pg_hba.conf`. We need to change the listen address, port, and also need to add a couple of lines for our zabbix_db database

```
listen_addresses = '*'
port = 5432
# configuration for Zabbix
local   zabbix_db   zabbix                               md5
host     zabbix_db   zabbix   <CIDR-address>             md5
```

The `local` keyword matches all the connections made in the Unix-domain sockets. This line is followed by the database name (`zabbix_db`), the username (`zabbix`), and the authentication method (in our case `md5`).

The `host` keyword matches all the connections that are coming from TCP/IP (this includes the SSL and non-SSL connections) followed by the database name (`zabbix_db`), the username (`zabbix`), the network and mask of all the hosts that are allowed, and the authentication method (in our case `md5`).

8. The network mask of the allowed hosts in our case should be a network mask because we need to allow the web interface (hosted on our web server) and the Zabbix server that is on a different dedicated server for example, `10.6.0.0/24` (a small subnet) or a large network as well. Most likely, the web interface as well the Zabbix server will be in a different network, so make sure you express all the networks and relative mask here.
9. Finally, we can start our PostgreSQL server using the following command:

```
# service postgresql-9.2 start
```

Or:

```
# /etc/init.d/postgresql-9.2 start
```

To create a database, we need to be a Postgres user (or the user that in your distribution is running PostgreSQL). Create a user for the database (our Zabbix user) and log in as that user to import the schema with the relative data.

The code to import the schema is as follows:

```
# su - postgres
-bash-4.1$
```

Once we become a postgres user, we can create the database (in our example, it is `zabbix_db`):

```
-bash-4.1$ psql
postgres=# CREATE USER zabbix WITH PASSWORD '<YOUR-ZABBIX-PASSWORD-
HERE>';
CREATE ROLE
postgres=# CREATE DATABASE zabbix_db WITH OWNER zabbix
ENCODING='UTF8';
CREATE DATABASE
postgres=# \q
```

The database creation scripts are located in the `./database/postgresql` folder. They should be installed exactly in this order:

```
# cat schema.sql | psql -U zabbix zabbix_db
# cat images.sql | psql -U zabbix zabbix_db
# cat data.sql | psql -U zabbix zabbix_db
```

Starting up with Daemon

Now, finally, it is the time to start our Zabbix server and test the whole setup for our Zabbix server/database.

```
# /etc/init.d/zabbix-server start
Starting Zabbix server: [ OK ]
```

A quick check on the logfile can give us more information about what is currently happening in our server. We should be able to get the following lines from the logfile (the default location is `/tmp/zabbix_server.log`):

```
28742:20130609:133955.418 Starting Zabbix Server. Zabbix 2.0.6
(revision 35158).
28742:20130609:133955.418 ***** Enabled features *****
28742:20130609:133955.418 SNMP monitoring: YES
28742:20130609:133955.418 IPMI monitoring: YES
28742:20130609:133955.418 WEB monitoring: YES
28742:20130609:133955.418 Jabber notifications: YES
28742:20130609:133955.418 Ez Texting notifications: YES
28742:20130609:133955.418 ODBC: NO
28742:20130609:133955.418 SSH2 support: YES
28742:20130609:133955.418 IPv6 support: YES
28742:20130609:133955.418 *****
```

The preceding lines tell us that our server is up and running. If the logfiles continue with the following lines, it means that is all fine: the configuration, the connection to database, and so on:

```
28745:20130609:133955.469 server #2 started [db watchdog #1]
28744:20130609:133955.471 server #1 started [configuration syncer #1]
28754:20130609:133955.494 server #9 started [trapper #1]
28755:20130609:133955.496 server #10 started [trapper #2]
28756:20130609:133955.498 server #11 started [trapper #3]
28757:20130609:133955.499 server #12 started [trapper #4]
28758:20130609:133955.501 server #13 started [trapper #5]
28763:20130609:133955.519 server #15 started [alerter #1]
28762:20130609:133955.526 server #14 started [icmp pinger #1]
28765:20130609:133955.566 server #17 started [timer #1]
28764:20130609:133955.574 server #16 started [housekeeper #1]
28764:20130609:133955.575 executing housekeeper
```

Another thing to check is whether our server is running with our user:

```
# ps -ef |grep zabbix_server
502 28742 1 0 13:39 ? 00:00:00 /usr/local/sbin/zabbix_server
502 28744 28742 0 13:39 ? 00:00:00 /usr/local/sbin/zabbix_server
502 28745 28742 0 13:39 ? 00:00:00 /usr/local/sbin/zabbix_server
...
```

The preceding lines show that `zabbix_server` is running with the user 502. We will go ahead and verify if 502 is the user we previously created:

```
# getent passwd 502
zabbixsvr:x:502:501::/home/zabbixsvr:/bin/bash
```

The preceding lines show that is all fine. The most common issue normally is the following error:

```
28487:20130609:133341.529 Database is down. Reconnecting in 10
seconds.
```

We can have different actors that are causing this issue:

- Firewall (local on our servers or infrastructure firewall)
- The Postgres configuration
- Wrong data on `zabbix_server.conf`



We can try to isolate the problem by running the following command on the database server:

```
serverpsql -h 127.0.0.1 -U zabbix zabbix_dbPassword for
user zabbix:psql (9.2.4) Type "help" for help
```

If we have a connection, we can try the same command from the Zabbix server; if it fails, it is better to check the firewall configuration. If we get the fatal identification-authentication failed error, it is better to check the `pg_hba.conf` file.

Now it is time to check how to start and stop your Zabbix installation. The scripts that follow are a bit customized to manage the different users for the server and the agent.



The following startup script works fine with the standard compilation without using a `--` prefix and the `zabbixsvr` user. If you are running on a different setup, make sure you customize the executable location and the user:

```
exec=/usr/local/sbin/zabbix_server
zabbixsrv=zabbixsvr
```

For `zabbix-server`, create the `zabbix-server` file at `/etc/init.d` with the following content:

```
#!/bin/sh
#
# chkconfig: - 85 15
# description: Zabbix server daemon
# config: /usr/local/etc/zabbix_server.conf
#

### BEGIN INIT INFO
# Provides: zabbix
# Required-Start: $local_fs $network
# Required-Stop: $local_fs $network
# Default-Start:
# Default-Stop: 0 1 2 3 4 5 6
# Short-Description: Start and stop Zabbix server
# Description: Zabbix server
### END INIT INFO

# Source function library.
. /etc/rc.d/init.d/functions

exec=/usr/local/sbin/zabbix_server
prog=${exec##*/}
lockfile=/var/lock/subsys/zabbix
syscf=zabbix-server
```

The next parameter `zabbixsvr` is specified inside the `start()` function and it determines which user will be used to run our Zabbix server:

```
zabbixsrv=zabbixsvr
[ -e /etc/sysconfig/$syscf ] && . /etc/sysconfig/$syscf

start()
{
    echo -n $"Starting Zabbix server: "
```

In the preceding code, the user (who will own our Zabbix's server process) is specified inside the `start` function.

```
daemon --user $zabbixsrv $exec
```

Remember to change the ownership of the server logfile and configuration file of Zabbix, this to prevent a normal user to access sensitive data that can be acquired with Zabbix. Logfile is specified as follows:

```
/usr/local/etc/zabbix_server.conf
On 'LogFile' 'LogFile' properties    rv=$?
    echo
    [ $rv -eq 0 ] && touch $lockfile
    return $rv
}

stop()
{
    echo -n "Shutting down Zabbix server: "
```

Here, inside the `stop` function, we don't need to specify the user as the `start/stop` script runs from root so we can simply use `killproc $prog` as follows:

```
    killproc $prog
    rv=$?
    echo
    [ $rv -eq 0 ] && rm -f $lockfile
    return $rv
}

restart()
{
    stop
    start
}

case "$1" in
    start|stop|restart)
        $1
        ;;
    force-reload)
        restart
        ;;
    status)
        status $prog
        ;;
    try-restart|condrestart)
        if status $prog >/dev/null ; then
            restart
        fi
    ;;
    *)
        echo "Usage: $0 {start|stop|restart|force-reload|status|try-restart|condrestart}"
        exit 1
    ;;
esac
```



```
        ;;
    reload)
        action $"Service ${0##*/} does not support the reload
        action: " /bin/false
        exit 3
        ;;
    *)
        echo $"Usage: $0 {start|stop|status|restart|try-restart|force-
        reload}"
        exit 2
        ;;
esac
```



The following startup script works fine with the standard compilation without using a --prefix and the zabbix_usr user. If you are running on a different setup, make sure you customize the executable location and the user:

```
exec=/usr/local/sbin/zabbix_agentd
zabbix_usr=zabbix
```

For zabbix_agent, create the following zabbix-agent file at /etc/init.d/zabbix-agent:

```
#!/bin/sh
#
# chkconfig: - 86 14
# description: Zabbix agent daemon
# processname: zabbix_agentd
# config: /usr/local/etc/zabbix_agentd.conf
#

### BEGIN INIT INFO
# Provides: zabbix-agent
# Required-Start: $local_fs $network
# Required-Stop: $local_fs $network
# Should-Start: zabbix zabbix-proxy
# Should-Stop: zabbix zabbix-proxy
# Default-Start:
# Default-Stop: 0 1 2 3 4 5 6
# Short-Description: Start and stop Zabbix agent
# Description: Zabbix agent
### END INIT INFO

# Source function library.
. /etc/rc.d/init.d/functions

exec=/usr/local/sbin/zabbix_agentd
```

```

prog=${exec##*/}
syscf=zabbix-agent
lockfile=/var/lock/subsys/zabbix-agent

```

The following `zabbix_usr` parameter specifies the account that will be used to run Zabbix's agent:

```

zabbix_usr=zabbix
[ -e /etc/sysconfig/$syscf ] && . /etc/sysconfig/$syscf

start()
{
    echo -n $"Starting Zabbix agent: "

```

The next command uses the value of the `zabbix_usr` variable and permits us to have two different users, one for server and one for agent, preventing the Zabbix agent from accessing the `zabbix_server.conf` file that contains our database password.

```

    daemon --user $zabbix_usr $exec
    rv=$?
    echo
    [ $rv -eq 0 ] && touch $lockfile
    return $rv
}

stop()
{
    echo -n $"Shutting down Zabbix agent: "
    killproc $prog
    rv=$?
    echo
    [ $rv -eq 0 ] && rm -f $lockfile
    return $rv
}

restart()
{
    stop
    start
}

case "$1" in
    start|stop|restart)
        $1
        ;;
    force-reload)
        restart
        ;;
    status)

```

```
        status $prog
        ;;
    try-restart|condrestart)
        if status $prog >/dev/null ; then
            restart
        fi
        ;;
    reload)
        action $"Service ${0##*/} does not support the reload action:
" /bin/false
        exit 3
        ;;
    *)
        echo $"Usage: $0 {start|stop|status|restart|try-restart|force-
reload}"
        exit 2
        ;;
esac
```

With that setup, we have the agent that is running with `zabbix_usr` and the server with Unix accounts of `zabbixsvr`.

```
zabbix_usr_ 4653 1 0 15:42 ?      00:00:00 /usr/local/sbin/zabbix_
agentd
zabbix_usr 4655 4653  0 15:42 ?    00:00:00 /usr/local/sbin/zabbix_
agentd
zabbixsvr 4443 1  0 15:32 ?      00:00:00 /usr/local/sbin/zabbix_server
zabbixsvr 4445 4443  0 15:32 ? 00:00:00 /usr/local/sbin/zabbix_server
```

Some considerations about the database


Zabbix uses an interesting way to keep the database the same size at all times. The database size indeed depends upon:

- Number of processed values per second
- Housekeeper settings

Zabbix uses two ways to store the collected data:


- History
- Trends

While on history, we will find all the collected data (it doesn't matter what type of data will be stored on history), trends will collect only numerical data. Its minimum, maximum, and average calculations are consolidated by hour (to keep the trend a lightweight process).

 All the "strings" items such as character, log, and text do not correspond to trends since trends store only values.

There is a process called the housekeeper that is responsible to handle the retention against our database. It is strongly advised to keep the data in history as small as possible so that we do not overload the database with a huge amount of data and store the trends as long as we want.

Now since Zabbix will also be used for capacity planning purposes, we need to consider having a baseline and keeping at least a whole business period. Normally, the minimum period is one year but it is strongly advised to keep the trend history on for at least two years. These historical trends will be used during the business opening and closure to have a baseline and quantify the overhead of a specified period.

 If we indicate 0 as the value for trends, the server will not calculate or store trends at all. If history is set to "0", Zabbix will be able to calculate only triggers based on the last value of the item itself as it does not store historical values at all.

The most common issue that we face when aggregating data is to have values influenced by some positive spikes or fast drops in our hourly based trends, this means that huge spikes can produce a mean value per hour that is not right.

Trends in Zabbix are implemented in a smart way. The script creation for the trend table is as follows:

```
CREATE TABLE trends(
  itemid bigint NOT NULL, clock integer DEFAULT '0'
  NOT NULL, num integer DEFAULT '0'
  NOT NULL, value_min numeric(16, 4) DEFAULT '0.0000'
  NOT NULL, value_avg numeric(16, 4) DEFAULT '0.0000'
  NOT NULL, value_max numeric(16, 4) DEFAULT '0.0000'
  NOT NULL, PRIMARY KEY(itemid, clock));

CREATE TABLE trends_uint(
  Itemid bigint NOT NULL, Clock integer DEFAULT '0'
  NOT NULL, Num integer DEFAULT '0'
  NOT NULL, value_min numeric(20) DEFAULT '0'
  NOT NULL, value_avg numeric(20) DEFAULT '0'
  NOT NULL, value_max numeric(20) DEFAULT '0'
  NOT NULL, PRIMARY KEY(itemid, clock));
```

As you can see, there are two tables showing trends inside the Zabbix database:

- Trends
- Trends_uint

The first table, Trends, is used to store the float value. The second table, trends_uint, is used to store the unsigned integer. Both tables own the concept of keeping for each hour the:

- Minimum value (value_min)
- Maximum value (value_max)
- Average value (value_avg)

This feature permits us to find out and display the trends graphically using the influence of spikes and fast drop against the average value and understanding how and how much this value has been influenced. The other tables used for the historical purpose are as follows:

- history: It is used to store numeric data (float)
- history_log: It is used to store logs (for example, the text field on PostgreSQL variable has unlimited length)
- history_str: It is used to store strings (up to 255 characters)
- history_text: It is used to store the text value (here again is a text field, so it has unlimited length)
- history_uint: It is used to store numeric values (unsigned integers)

Sizing of the database

Calculating the definitive database size is not an easy task because it is hard to predict how many items and the relative rate per second we will have on our infrastructure and how many events will be generated. To simplify this, we will consider the worst scenario where we have an event generated per second.

Summarizing the database size is influenced by:

- **Items:** The number of items in particular
- **Refresh rate:** The average refresh rate of our items
- **Space to store values:** This value depends on RDBMS

The space used to store the data may vary from database to database, but we can simplify our work by considering some mean values to have a value that quantifies the maximum space consumed by the database. We can also consider the space used to store values on history around 50 bytes per value, the space used from a value on the trend table is normally around 128 bytes and the space used for a single event is normally around 130 bytes.

The total amount of used space can be calculated with the following formula:

Configuration + History + Trends + Events

Now, let us look into each of the components:

- **Configuration:** This refers to the Zabbix's configuration for the server, the web interface, and all the configuration parameters that are stored into the database; this occupation is normally around 10 MB.
- **History:** The history component is calculated using the following formula:

$$\text{History retention days} * (\text{items/refresh rate}) * 24 * 3600 * 50 \text{ bytes}$$
 (History bytes usage average)
- **Trends:** The trends component is calculated using the following formula:

$$\text{days} * (\text{items}/3600) * 24 * 3600 * 128 \text{ bytes}$$
 (Trend bytes usage average)
- **Events:** The event component is calculated using the following formula:

$$\text{days} * \text{events} * 24 * 3600 * 130 \text{ bytes}$$
 (Event bytes usage average)

Now coming back to our practical example, we can consider having 5000 items refreshed every minute and we would like to have retention of 7 days the used space will be calculated as follows:

History: retention (in days) * (items/refresh rate) * 24 * 3600 * 50 bytes

50 bytes is the mean value for the space consumed by a value stored on history.

Considering 30 days the result is the following:


- History will be calculated as:

$$30 * 5000 / 60 * 24 * 3600 * 50 = 10.8 \text{GB}$$
- As we said earlier, to simplify, we will consider the worst scenario (one event per second) and will also consider keeping 5 years of events

- Events will be calculated using the following formula:
`retention days*events*24*3600* Event bytes usage (average)`

When we calculate an event we have:


$$5*365*24*3600* 130 = 15.7GB$$

 130 bytes is the mean value for the space consumed for a value stored on events.

- Trends will be calculated using the following formula:
`retention in days*(items/3600)*24*3600*Trend bytes usage (average)`

When we calculate trends we have:

$$5000*24*365* 128 = 5.3GB \text{ per year or } 26.7GB \text{ for 5 years.}$$

 128 bytes is the mean value for the space consumed for a value stored on trends.

The following table shows the retention in days and the space required for the measure:

Type of measure	Retention in days	Space required
History	30	10.8 GB
Events	1825 (5 years)	15.7 GB
Trends	1825 (5 years)	26.7 GB
Total	N.A.	53.2 GB

The calculated size is not the initial size of our database but we need to keep in mind that this one will be our target size after 5 years. We are also considering a history of 30 days, keep in mind that this retention can be reduced in case of issues since the trends will keep and store our baseline and hourly based trends.

The history and trends retention policy can be changed easily per item. This means we can create a template with items that have a different history retention by default. Normally, the history is set to 7 days, but for some kind of measure such as in a web scenario, or other particular measure, we may need to keep all the values for more than a week. This permits us to change the value of each item.

In our example we performed a worst-case scenario with retention of 30 days, but it is good advice to keep the history only for 7 days or maybe less on large environments. If we perform a basic calculation of an item that is updated every 60 seconds and has history preserved for 7 days, it will generate:

(update interval) * (hours in a day) * (number of days in history) = $60 * 24 * 7 = 10080$

This means that for each item we will have 10,080 lines in a week and that gives us an idea of the number of lines that we will produce on our database.

The screenshot shows the Zabbix Item configuration form. The item is named 'CPU \$2 time' and is of type 'Zabbix agent'. The key is 'system.cpu.util[,iowait]'. The update interval is set to 60 seconds. The 'Keep history (in days)' field is highlighted with a red box and contains the value 7. The 'Keep trends (in days)' field contains the value 365. The form also includes fields for parent items, host, name, type, key, host interface, type of information, units, and a section for flexible intervals.

Interval	Period	Action
No flexible intervals defined.		

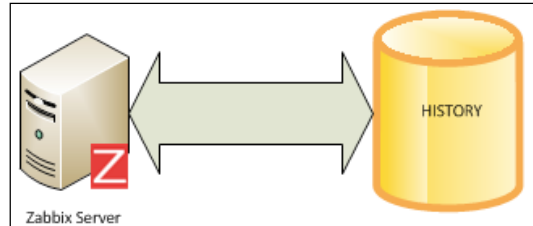
New flexible interval	Interval (in sec)	Period	Action
	50	1-7,00:00-24:00	Add

Some considerations about housekeeping

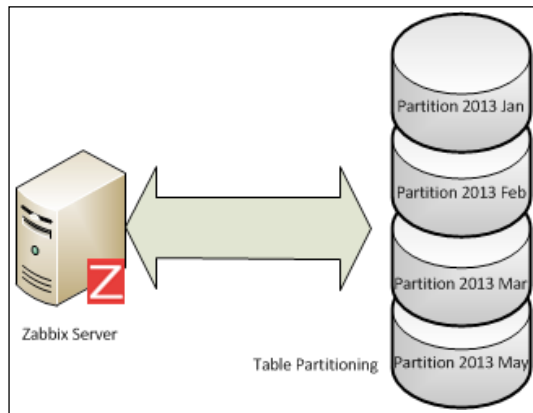
Housekeeping can be quite a heavy process. As the database grows, housekeeping will require more and more time to complete its work. This issue can be sorted using the `delete_history()` function.

There is a way to deeply improve the housekeeping performance and fix this performance drop. The heaviest tables are: `history`, `history_uint`, `trends`, and `trends_uint`.

A solution can be the PostgreSQL table partitioning and partition of the entire tables on a monthly basis.



The preceding figure displays the standard and nonpartitioned `history` table on the database. The following figure shows how a partitioned `history` table will be stored in the database.



Partitioning is basically the splitting of a logical large table into smaller physical pieces. This feature can provide several benefits:

- Performance of queries can be improved dramatically in situations when there is a heavy access to the table's rows in a single partition
- The partitioning will reduce the index size making it more likely to fit in memory of the heavy parts that are being used
- Massive deletes can be accomplished by removing partitions, instantly reducing the space allocated for the database without introducing fragmentation and heavy load on index rebuilding. The `delete partition` command also entirely avoids the *vacuum* overhead caused by a bulk *delete*
- When a query updates or requires the access to a large percentage of the partition, using the sequential scan is often more efficient than the index usage with a random access or scattered reads against that index

All these benefits are worthwhile only when a table is very large. The strong point of this kind of architecture is that the RDBMS will directly access the needed partition, and the delete will simply be a delete of a partition. Partition deletion is a fast process and requires little resource.

Unfortunately, Zabbix is not able to manage the partitions so we need to disable the housekeeping and use an external process to accomplish the housekeeping.

To set up this feature, we need to create a schema where we can place all the partitioned tables:

```
CREATE SCHEMA partitions AUTHORIZATION zabbix;
```

Now we need a function that will create the partition. So to connect to Zabbix you need to run the following code:

```
CREATE OR REPLACE FUNCTION trg_partition()
RETURNS TRIGGER AS
$BODY$
DECLARE
    prefix text := 'partitions.';
    timeformat text;
    selector text;
    _interval INTERVAL;
    tablename text;
    startdate text;
    enddate text;
    create_table_part text;
    create_index_part text;
BEGIN
    selector = TG_ARGV[0];
    IF selector = 'day'
    THEN
        timeformat := 'YYYY_MM_DD';
    ELSIF selector = 'month'
    THEN
        timeformat := 'YYYY_MM';
    END IF;

    _interval := '1 ' || selector;
    tablename := TG_TABLE_NAME || '_p' || TO_CHAR(TO_TIMESTAMP(NEW.clock),
timeformat);

    EXECUTE 'INSERT INTO ' || prefix || quote_ident(tablename) || ' SELECT
($1).*'
    USING NEW;
    RETURN NULL;

EXCEPTION
```

```
    WHEN undefined_table THEN

    startdate := EXTRACT(epoch FROM date_trunc(selector, TO_TIMESTAMP(NEW.
    clock)));
    enddate := EXTRACT(epoch FROM date_trunc(selector, TO_TIMESTAMP(NEW.
    clock) + _interval));

    create_table_part := 'CREATE TABLE IF NOT EXISTS ' || prefix || quote_
    ident(tablename) || ' (CHECK ((clock >= ' || quote_literal(startdate)
    || ' AND clock < ' || quote_literal(enddate) || ')) INHERITS (' ||
    TG_TABLE_NAME || '));
    create_index_part := 'CREATE INDEX ' || quote_ident(tablename) || '_1
    on ' || prefix || quote_ident(tablename) || '(itemid,clock)';

    EXECUTE create_table_part;
    EXECUTE create_index_part;

    --insert it again
    EXECUTE 'INSERT INTO ' || prefix || quote_ident(tablename) || ' SELECT
    ($1).*'
    USING NEW;
    RETURN NULL;

END;
$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;
ALTER FUNCTION trg_partition()
OWNER TO zabbix;
```

Now we need a trigger connected to each table that we want to separate. This trigger will run an INSERT statement and if the partition is not ready or yet created, the function will create the partition right before the INSERT statement.

```
CREATE TRIGGER partition_trg BEFORE INSERT ON history
FOR EACH ROW EXECUTE PROCEDURE trg_partition('day');
CREATE TRIGGER partition_trg BEFORE INSERT ON history_sync
FOR EACH ROW EXECUTE PROCEDURE trg_partition('day');
CREATE TRIGGER partition_trg BEFORE INSERT ON history_uint
FOR EACH ROW EXECUTE PROCEDURE trg_partition('day');
CREATE TRIGGER partition_trg BEFORE INSERT ON history_str_sync
FOR EACH ROW EXECUTE PROCEDURE trg_partition('day');
CREATE TRIGGER partition_trg BEFORE INSERT ON history_log
FOR EACH ROW EXECUTE PROCEDURE trg_partition('day');
CREATE TRIGGER partition_trg BEFORE INSERT ON trends
FOR EACH ROW EXECUTE PROCEDURE trg_partition('month');
CREATE TRIGGER partition_trg BEFORE INSERT ON trends_uint
FOR EACH ROW EXECUTE PROCEDURE trg_partition('month');
```

At this point, we miss only the housekeeping function that will replace the one built in Zabbix and disable Zabbix's native one.

The function that will handle housekeeping for us is as follows:

```
CREATE OR REPLACE FUNCTION delete_partitions(intervaltodelete
INTERVAL, tabletype text)
RETURNS text AS
$BODY$
DECLARE
result RECORD ;
prefix text := 'partitions.';
table_timestamp TIMESTAMP;
delete_before_date DATE;
tablename text;
BEGIN
    FOR result IN SELECT * FROM pg_tables WHERE schemaname =
'partitions' LOOP
        table_timestamp := TO_TIMESTAMP(substring(result.tablename
FROM '[0-9_]*$', 'YYYY_MM_DD');
        delete_before_date := date_trunc('day', NOW() -
intervalToDelete);
        tablename := result.tablename;
        IF tabletype != 'month' AND tabletype != 'day' THEN
            RAISE EXCEPTION 'Please specify "month" or "day" instead of %',
tabletype;
        END IF;
        --Check whether the table name has a day (YYYY_MM_DD) or month
(YYYY_MM) format
        IF LENGTH(substring(result.tablename FROM '[0-9_]*$')) = 10
AND tabletype = 'month' THEN
            --This is a daily partition YYYY_MM_DD
            -- RAISE NOTICE 'Skipping table % when trying
to delete "%" partitions (%)', result.tablename, tabletype,
length(substring(result.tablename from '[0-9_]*$'));
            CONTINUE;
        ELSIF LENGTH(substring(result.tablename FROM '[0-9_]*$')) = 7
AND tabletype = 'day' THEN
            --this is a monthly partition
            --RAISE NOTICE 'Skipping table % when trying to delete "%"
partitions (%)', result.tablename, tabletype, length(substring(result.
tablename from '[0-9_]*$'));
            CONTINUE;
        ELSE
            --This is the correct table type. Go ahead and check if it
needs to be deleted
            --RAISE NOTICE 'Checking table %', result.tablename;
            END IF;
    END LOOP;
END;
```

```
IF table_timestamp <= delete_before_date THEN
    RAISE NOTICE 'Deleting table %', quote_ident(tablename);
    EXECUTE 'DROP TABLE ' || prefix || quote_ident(tablename) ||
';';
END IF;
END LOOP;
RETURN 'OK';
END;
$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;
ALTER FUNCTION delete_partitions(INTERVAL, text)
OWNER TO zabbix;
```

Now you have the housekeeping ready to run. To enable housekeeping, we can use the crontab by adding the following entries:

```
@daily psql -q -U zabbix -c "SELECT delete_partitions('7 days',
'day') "
@daily psql -q -U zabbix -c "SELECT delete_partitions('24 months',
'month') "
```

Those two tasks should be scheduled on the database server's crontab. In this example we are keeping the history of 7 days and trends of 24 months.

Now we can finally disable the Zabbix housekeeping, using the following code:

```
### Option: DisableHousekeeping
# If set to 1, disables housekeeping.
#
# Mandatory: no
# Range: 0-1
DisableHousekeeping=1
```

Restart Zabbix, now the built-in housekeeping is disabled and you should see a lot of improvement in the performance. To keep your database as lightweight as possible, you can clean up the following tables:

- acknowledges
- alerts
- auditlog
- events
- service_alarms

Once you chose your own retention, you need to add a retention policy, for example, in our case it will be 2 years of retention. With the following crontab entries, you can delete all the records older than 63072000 (2 years expressed in seconds):

```
@daily psql -q -U zabbix -c "delete from acknowledges where clock <
(SELECT (EXTRACT( epoch FROM now() ) - 63072000))"
@daily psql -q -U zabbix -c "delete from alerts where clock < (SELECT
(EXTRACT( epoch FROM now() ) - 63072000))"
@daily psql -q -U zabbix -c "delete from auditlog where clock <
(SELECT (EXTRACT( epoch FROM now() ) - 62208000))"
@daily psql -q -U zabbix -c "delete from events where clock < (SELECT
(EXTRACT( epoch FROM now() ) - 62208000))"
@daily psql -q -U zabbix -c "delete from service_alarms where clock <
(SELECT (EXTRACT( epoch FROM now() ) - 62208000))"
```

To disable housekeeping, we need to drop the triggers created:

```
DROP TRIGGER partition_trg ON history;
DROP TRIGGER partition_trg ON history_sync;
DROP TRIGGER partition_trg ON history_uint;
DROP TRIGGER partition_trg ON history_str_sync;
DROP TRIGGER partition_trg ON history_log;
DROP TRIGGER partition_trg ON trends;
DROP TRIGGER partition_trg ON trends_uint;
```

All those changes need to be tested and changed/modified as they fit your setup. Also be careful and back up your database.

The web interface

The web interface installation is quite easy there are some basic steps to execute. The web interface is completely written on PHP so we need a web server that supports PHP, in our case, we will use Apache with the PHP support enabled.

The entire web interface is contained inside the php folder at `frontends/php/` that we need to copy on our `htdocs` folder:

```
/var/www/html
```

Use the following commands to copy the folders:

```
# mkdir <htdocs>/zabbix
# cd frontends/php
# cp -a . <htdocs>/zabbix
```



Be careful, you might need proper rights and permissions as all those files are owned by Apache and they also depend on your httpd configuration.

The web wizard – frontend configuration

Now, from your web browser you need to open the following URL:

```
http://<server_ip_or_name>/zabbix
```

The first screen that you will meet is a welcome page; there is nothing to do there other than to click on **Next**. When on the first page, you may get a warning on your browser that informs you that the date/timezone is not set. This is a parameter inside the `php.ini` file. All the possible timezones are described on the official PHP website:

```
http://www.php.net/manual/en/timezones.php
```

The parameter to change is the date/timezone inside the `php.ini` file. If you don't know the current PHP configuration or where it is located in your `php.ini` file, and you need some detailed information about which modules are running or the current settings then you can write a file, for example, `php-info.php` inside the Zabbix directory with the following content:

```
<?phpphpinfo();phpinfo(INFO_MODULES);
?>
```

Now point your browser to:

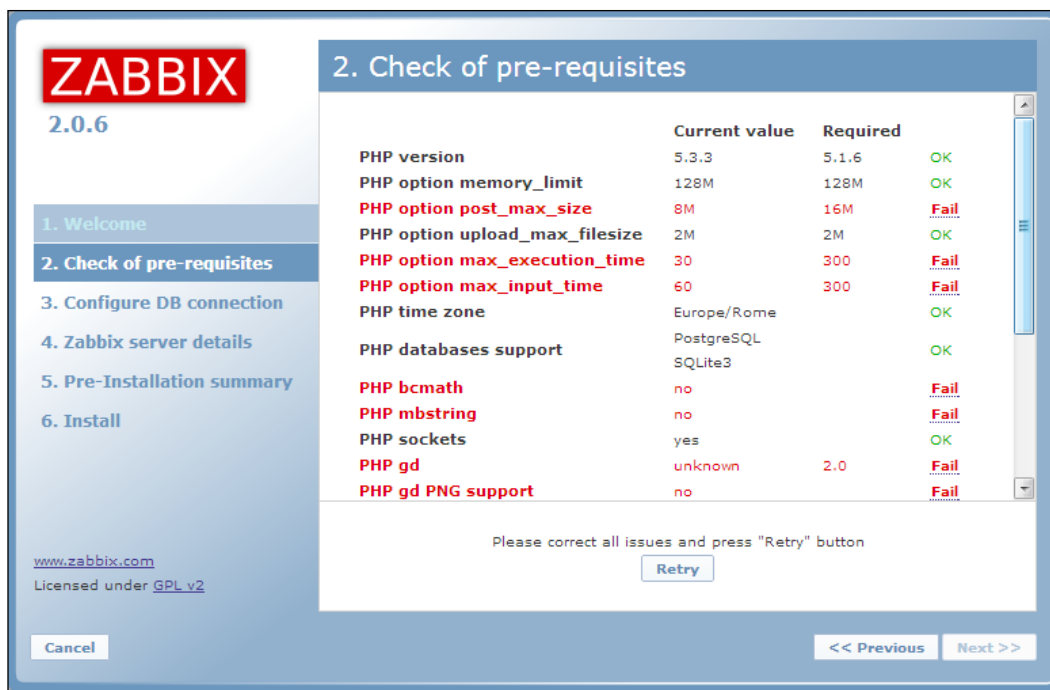
```
http://your-zabbix-web-frontend/zabbix/php-info.php
```

You will have your full configuration printed out in a webpage. The next screenshot is more important, it displays a prerequisites check and as you can see there are three prerequisites that are not met:



Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.



On standard Red-Hat/CentOS, it is easy to find the prerequisites that are not fulfilled:

PHP option post_max_size	8M	16M	Fail
PHP option max_execution_time	30	300	Fail
PHP option max_input_time	60	300	Fail
PHP bcmath	no		Fail
PHP mbstring	no		Fail
PHP gd unknown		2.0	Fail
PHP gd PNG support	no		Fail
PHP gd JPEG support	no		Fail
PHP gd FreeType support	no		Fail
PHP xmlwriter	no		Fail
PHP xmlreader	no		Fail

Most of these parameters are contained inside the `php.ini` file. To fix them, I have changed the following options inside the `/etc/php.ini` file:

```
[Date]
; Defines the default timezone used by the date functions
; http://www.php.net/manual/en/datetime.configuration.php#ini.date.
timezone
date.timezone = Europe/Rome

; Maximum size of POST data that PHP will accept.
; http://www.php.net/manual/en/ini.core.php#ini.post-max-size
post_max_size = 16M

; Maximum execution time of each script, in seconds
; http://www.php.net/manual/en/info.configuration.php#ini.max-
execution-time
max_execution_time = 300

; Maximum amount of time each script may spend parsing request data.
It's a good
; idea to limit this time on productions servers in order to eliminate
unexpectedly
; long running scripts.
; Default Value: -1 (Unlimited)
; Development Value: 60 (60 seconds)
; Production Value: 60 (60 seconds)
; http://www.php.net/manual/en/info.configuration.php#ini.max-input-
time
max_input_time = 300

; Maximum amount of time each script may spend parsing request data.
It's a good
; idea to limit this time on productions servers in order to eliminate
unexpectedly
; long running scripts.
; Default Value: -1 (Unlimited)
; Development Value: 60 (60 seconds)
; Production Value: 60 (60 seconds)
; http://www.php.net/manual/en/info.configuration.php#ini.max-input-
time
max_input_time = 300
```

To solve the missing library, we need to install the following packages:

- php-xml
- php-bcmath
- php-mbstring
- php-gd

We will use the following command to install these packages:

```
# yum install php-xml php-bcmath php-mbstring php-gd
```

The whole list or the prerequisite list is given in the following table:

Prerequisite	Min value	Solution
PHP Version	5.1.6	
PHP memory_limit	128M	In php.ini, change memory_limit=128M.
PHP post_max_size	16M	In php.ini, change post_max_size=16M.
PHP upload_max_filesize	2M	In php.ini, change upload_max_filesize=2M.
PHP max_execution_time option	300 Seconds	In php.ini, change max_execution_time=300.
PHP max_input_time option	300 seconds	In php.ini, change max_input_time=300.
PHP session.auto_start	Disabled	In php.ini, change the session.auto_start=0.
bcmath		Use php-bcmath extension
mbstring		Use php-mbstring extension
sockets		This extension is required for the user script support php-net-socket module
gd		PHP GD extension must support PNG images (--with-png-dir), JPEG (--with-jpeg-dir) images, and FreeType 2 (--with-freetype-dir)

Prerequisite	Min value	Solution
libxml		Use php-xml or php5-dom
xmlwriter		Use php-xmlwriter
xmlreader		Use php-xmlreader
ctype		Use php-ctype
session		Use php-session
gettext		Use php-gettext

Every time you change a `php.ini` file or install a PHP extension the `httpd` needs a restart to get the change. Once all the prerequisites are met, we can click on **Next** and go ahead. On the next screen, we need to configure the database connection. We simply need to fill the form with the username, password, IP address, or hostname and also need to specify the kind of database server we are using.

ZABBIX
2.0.6

1. Welcome
2. Check of pre-requisites
3. Configure DB connection
4. Zabbix server details
5. Pre-Installation summary
6. Install

www.zabbix.com

Licensed under [GPL v2](#)

Cancel

<< Previous

Next >>

3. Configure DB connection

Please create database manually,
and set the configuration parameters for connection to this database.

Press "Test connection" button when done.

Database type

PostgreSQL

Database host

zabbix_dbsvr

Database port

0 0 - use default port

Database name

zabbix_db

User

zabbix

Password

OK

Test connection

If the connection is fine (can be checked with a test connection), we can proceed to the next step.

The screenshot shows the Zabbix 2.0.6 installation wizard interface. On the left is a sidebar with a navigation menu containing six items: '1. Welcome', '2. Check of pre-requisites', '3. Configure DB connection', '4. Zabbix server details' (which is highlighted with a blue background), '5. Pre-Installation summary', and '6. Install'. Below the menu, the sidebar includes the Zabbix logo, the version '2.0.6', the website 'www.zabbix.com', and the license 'Licensed under GPL v2'. At the bottom of the sidebar is a 'Cancel' button. The main content area is titled '4. Zabbix server details' and contains the following text: 'Please enter host name or host IP address and port number of Zabbix server, as well as the name of the installation (optional)'. Below this text are three input fields: 'Host' with the value 'zabbix_server', 'Port' with the value '10051', and 'Name' with the value 'zabbixserver'. At the bottom right of the main area are two buttons: '<< Previous' and 'Next >>'. The entire window has a blue border.

There is no check for connection available on this page so it is better to verify that it is possible to reach the Zabbix server from the network. In this form, it is necessary to fill the Host (or IP address) of our Zabbix server. Since we are installing the infrastructure on three different servers, we need to specify all the parameters and check that the Zabbix server port is available on the outside of the server.

Once we fill this form, we can click on **Next**. After this, the installation wizard proposes that we to view **Pre-Installation summary**, which is a complete summary of all the configuration parameters. If all is fine just click on **Next**, otherwise we can go back and change our parameters. When we go ahead, we see the configuration file has been generated (for example, in this installation in `//usr/share/zabbix/conf/zabbix.conf.php`).

It can happen that you may get an error instead of a success notification, and most probably, it is about the directory permission on our conf directory at `/usr/share/zabbix/conf`. Remember to make the directory writable to the httpd user (normally Apache is writable) at least for the time needed to create this file. Once this step is completed, the frontend is ready and we can perform our first login.

Capacity planning with Zabbix

Quite often people mix up the difference between capacity planning and performance tuning. Well the scope of performance tuning is to optimize the system you already have in place for better performance. Using your current performance acquired as a baseline, capacity planning determines what your system needs and when it is needed. Here, we will see how to organize our monitoring infrastructure to achieve this goal and provide us a useful baseline. Unfortunately, this chapter cannot cover all the aspects of this argument, we should have one whole book about capacity planning, but after this section, we will look at our Zabbix with a different vision and will be aware of what to do with it.

The observer effect

Zabbix is a good monitoring system because it is really lightweight. Unfortunately, every observed system will spend a bit of its resources to run the agent that acquires and measures data and metrics against the operating system so it is normal if the agent introduces a small (normally very small) overhead on the guest system. This is known as the **observer effect**. We can only accept this burden on our server and be aware that this will introduce a slight distortion in data collection, bearing in mind to keep it lightweight to a feasible extent while monitoring the process and our custom checks.

What to monitor

The Zabbix agent's job is to collect data periodically from the monitored machine and send metrics to the Zabbix server (that will be our aggregation and elaboration server). Now, in this scenario there are some important things to consider:


- What we are going to acquire?
- How we are going to acquire these metrics (the way or method used)?
- The frequency with which this measurement is performed

Considering the first point, it is important to think what should be monitored on our host and the kind of work that our host will do, or in other words what it will serve.

There are some basic metrics of operating systems that are nowadays more or less standardized, and those are: the CPU workload, percentage of free memory, memory usage details, usage of swap, CPU time for some process, and all this family of measure, all of them are built-in on the Zabbix agent.

Having a set of items with built-in measurement means that those are optimized to produce less workload possible on the monitored host, the whole of Zabbix's agent code is written in this way.

All the others metrics can be divided by service that our server should provide.

 Here templates are really useful! (Also, it is an efficient way to aggregate our metrics by type.)

Doing a practical example and considering monitoring an RDBMS it will be fundamental to acquire:

- All the operating system metrics
- Different custom RDBMS metrics

Our different custom RDBMS metrics can be: the number of users connected, or the usage of caches systems, the number of full table scans, and so on.

All those kind of metrics will be really useful and can be easily interpolated and compared against the same time period in a graph. Graphs have some strong points:

- Useful to understand (also from the business side)
- Often nice to present and integrate on slides to enforce our speech

Coming back to our practical example, well, currently we are acquiring data from our RDBMS and our operating system. (we can compare the workload of our RDBMS and see how this reflects the workload against our OS.) Well and now?

Most probably, our core business is the revenue of a website, merchant site, or some web application. We assume that we need to keep a website in a three-tier environment under control because it is a quite common case. Our infrastructure will be composed of the following actors:

- A web server
- An application server
- RDBMS

In real life, most probably this is the kind of environment Zabbix will be configured on. We need to be aware that every piece and every component that can influence our service should be measured and stored inside our Zabbix monitoring system. Generally we can consider it to be quite normal to see people with a strong system administration background to be more focused on operating system-related items as well. We also saw people writing Java code that needs to be concentrated on some other obscure measure such as the number of threads. The same kind of reasoning can be done if the capacity planner talks with a database administrator, or the specific guy of every sector.

This is a quite important point because the Zabbix implementer should have a global vision and should remember that when buying a new hardware the interface will most likely be a business unit.

This business unit very often doesn't know anything about the *number of threads* that our system can support, but will only understand customer satisfaction, customer-related issues, or how many concurrent users we can successfully serve.

Having said that, it is really important to be ready to talk in their language, and we can do that only if we have some efficient items to graph.

Define a baseline

Now if we look at the whole infrastructure from a client's point of view, we can think that if all our pages are served in a reasonable time the browsing experience will be pleasant.

Our goal in this case is to make our clients happy and the whole infrastructure reliable. Now we need to have two kinds of measures:

- The one felt from the user side (response time of our web pages)
- Infrastructure items related with it

We need to quantify the response time related to the user's navigation feeling, we need to know how much an user can wait in front of a web page to get a response, keeping in mind that the whole browsing experience needs to be pleasant. We can measure and categorize our metric with these three levels of response time:

- **0.2 seconds:** It gives the feel of an instantaneous response. The user feels the browser reaction was caused by him/her and not from a server with a business logic.

- **1-2 seconds:** The users feel the browsing is continuous without any interruption. The user can move freely rather than waiting for the pages to load.
- **10 seconds:** The liking for our website will drop. The user will want a better performance and can definitely be distracted by other things.

Now we have our thresholds and we can measure the response of a web page during normal browsing, in the meantime we can set some trigger level to warn us when the response time is more than two seconds for a page.

Now we need to relate that with all our other measures: the number of users connected, the number of sessions in our application server, and the number of connections to our database and relate all our measures with the response time and the number of users connected. Now we need to measure how our system is serving pages to users during normal browsing.

This can be defined as a baseline. It is where we currently are and is a measure of how our system is performing under a normal load.

Load testing

Now that we know how we are, and we have defined some threshold for our goal along with the pleasant browsing experience let's move forward.

We need to know which one is our limit, and more importantly how the system should reply to our requests. Since we can't hire a room full of people that can click on our website like crazy, we need to use the software to simulate this kind of behavior. There are some interesting open source software that do exactly this. There are different alternatives to choose from, one of them is Siege (<http://www.joedog.org/JoeDog/Siege>).

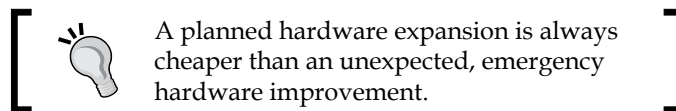
Seige permits us to simulate a stored browser history and load it on our server. We need to keep in mind that users, real users, will never be synchronized with each other. So it is important to introduce some delay between all the requests. Remember that if we have a login then we need to use a database of users because application servers cache their object and we don't want to measure how good the process is caching in them. The basic rule is to create a real browsing scenario against our website, so users who login can logout with just a click and without any random delay.

The stored scenarios should be repeated x times with a growing number of users, meaning our Zabbix will store our metrics, and at a certain point we will pass our first threshold (1-2 seconds per web page). We can go ahead until the response time reaches the value of our second threshold. There is no way to see how much load our server can take but it is well known that appetite comes with eating, so I will not be surprised if you go ahead and load your server until it crashes one of the components of your infrastructure.

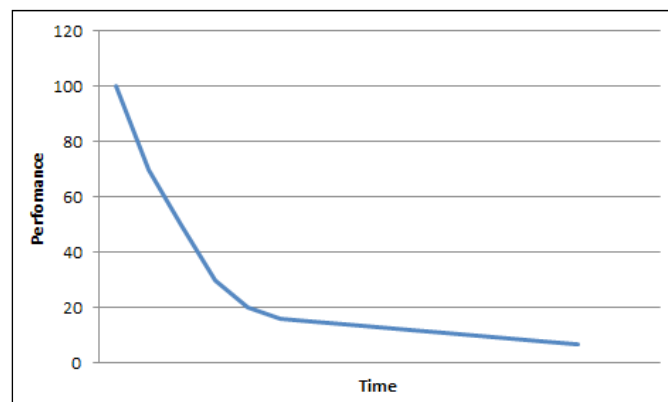
Drawing graphs that relate the response time with the number of users' server will help us see whether our three-tier web architecture is linear or not. Most probably it will grow in a linear pattern until a certain point. This segment is the one on which our system is performing fine. We can also see the components inside Zabbix and from this point we can introduce some kind of delay and draw some considerations.

Now we know exactly what to expect from our system and how the system can serve our users. We can see which component is the first that suffers the load, and where we need to plan a tuning.

Capacity planning can be done without digging and going deep into what to optimize. As said earlier, there are two different tasks: performance tuning and capacity planning that are related but different. We can simply review our performance and plan our infrastructure expansion.




We can also follow performance tuning, but be aware that there is a relation between the time spent and the performance obtained, so we need to understand when it is time to stop our performance tuning.



Forecasting the trends


One of the most important features of Zabbix is the capacity to store historical data. This feature is of vital importance during the task of predicting trends. Predicting our trends is not an easy task and is important considering the business that we are serving, and when looking at historical data we should see if there are repetitive periods or if there is a sort of formula that can express our trend.

For instance, it is possible that the online web store we are monitoring needs more and more resources during a particular period of the year, for example, close to public holidays if we sell travels. Doing a practical example, you can consider the used space on a specific server disk. Zabbix gives us the export functionality to get our historical data, so it is quite easy to import them in a spreadsheet. Excel has a curve fitting option that will help us a lot. It is quite easy to find a trend line using Excel that will tell us when we are going to reach all our disk space. To add a trend line into Excel, we need to create at first a "scattered graph" with our data; here it is also important to graph the disk size. After this we can try to find a mathematical equation that is closer to our trend. There are different kinds of formula that we can choose; in this example I used a Linear equation because the graphs are growing with a linear relation.

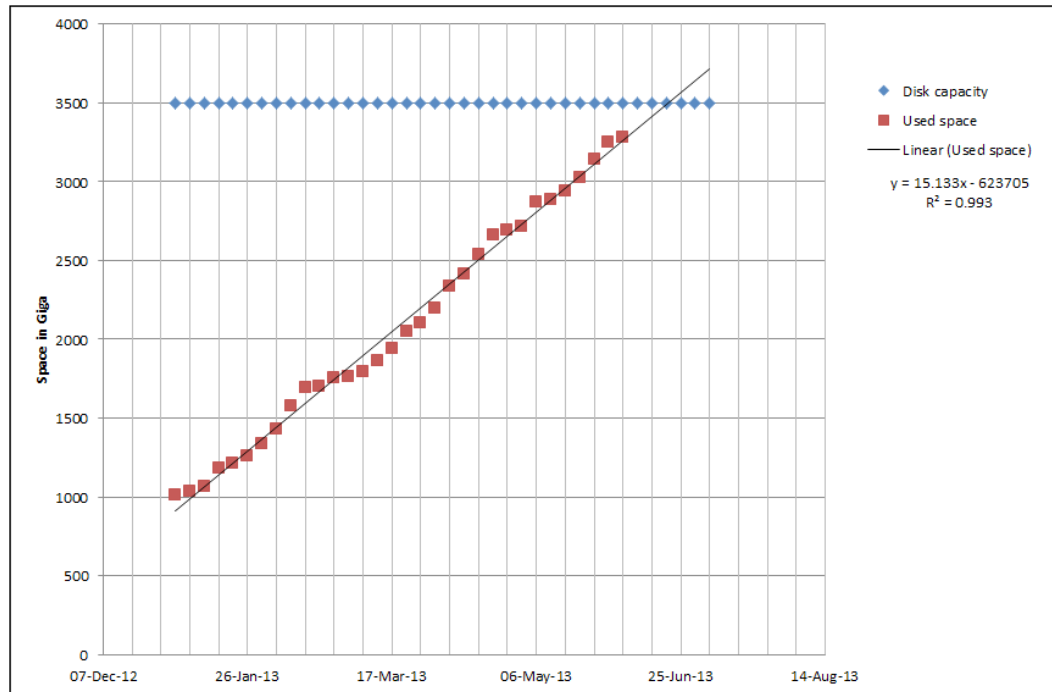
 The trend line process is also known as a **curve fitting process**.

The graph that comes out from this process permits us to know, with a considerable degree of precision, when we will run out of space.

Now it is clear how important it is to have a considerable quantity of historical data, bearing in mind the business period and how they influence data.

 It is important to keep a track of the trend/regression line used and the relative formula with R-squared value so that it is possible to calculate it with precision, and if there aren't any changes in trends, when the space will be exhausted.

The graph obtained is shown in the following figure, and from this graph it is simple to see that if the trends don't change, we are going to run out of space on June 25, 2013.



Summary

In this chapter, we completed a Zabbix setup in a three-tier environment. This environment is a good starting point to handle all the events generated from a large and a very large environment.

In next chapter, you will go deep into nodes, proxies, and all the possible infrastructure evolution and, as you will see, all of them are an improvement on this initial setup. This does not mean that the extensions described in the next chapter are easy to implement, but all the infrastructural improvements use this three-tier setup as a starting point. Basically, in the next chapter, you will learn how to expand and evolve this setup and also see how the distributed scenarios can be integrated in our installation. The next chapter will also include an important discussion about security in a distributed environment, making you aware of possible security risks that may arise on distributed environments.

2

Distributed Monitoring

Zabbix is a fairly lightweight monitoring application that is able to manage thousands of items with a single-server installation. However, the presence of thousands of monitored hosts, a complex network topology, or the necessity to manage different geographical locations with intermittent, slow, or faulty communications can all show the limits of a single-server configuration. Likewise, the necessity to move beyond a monolithic scenario towards a distributed one is not necessarily a matter of raw performance, and therefore, it's not just a simple matter of deciding between buying many smaller machines or just one big powerful one. Many DMZs and network segments with a strict security policy don't allow two-way communication between any hosts on either side, so it can be impossible for a Zabbix server to communicate with all the agents on the other side of a firewall. Different branches in the same company or different companies in the same group may need some sort of independence in managing their respective networks, while also needing some coordination and higher-level aggregation of monitored data. Different labs of a research facility may find themselves without a reliable network connection, so they may need to retain monitored data for a while and then send it asynchronously for further processing.

Thanks to its distributed monitoring features, Zabbix can thrive in all these scenarios and provide adequate solutions whether the problem is about performance, network segregation, administrative independence, or data retention in the presence of faulty links.

While the judicious use of the Zabbix agents could be considered from some point of view as a simple form of distributed monitoring, in this chapter, we will concentrate on Zabbix's two supported distributed monitoring modes: proxies and nodes. In this chapter, you will learn the differences between proxies and nodes, their respective advantages and disadvantages, how to deploy and configure them, and how to mix and match the two for really complex scenarios.

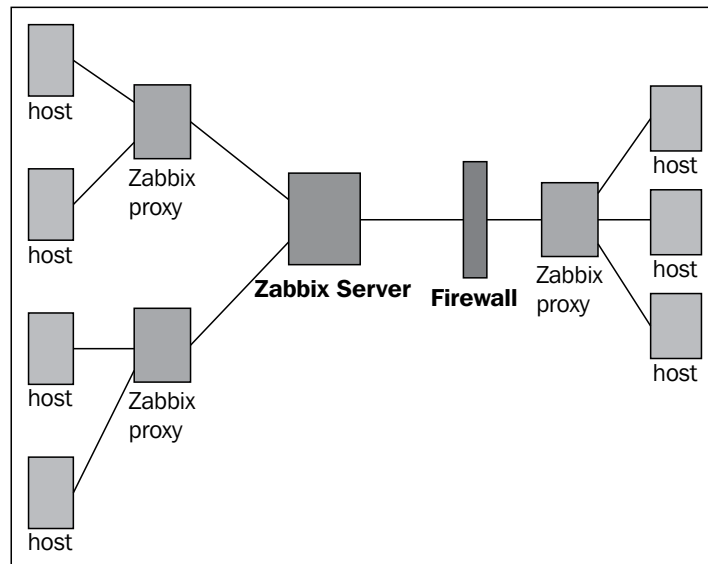
There will also be some considerations about security between proxies and nodes, so that by the end of this chapter, you will have all the information you need to apply Zabbix's distributed features to your environment.

Zabbix proxies

A Zabbix proxy is another member of the Zabbix suite of programs that sits between a full blown Zabbix server and a host-oriented Zabbix agent. Just like a server, it's used to collect data from any number of items on any number of hosts, and it can retain that data for an arbitrary period of time, relying on a dedicated database to do so. Just like an agent, it doesn't have a frontend and is managed directly from the central server. It also limits itself to data collections without triggering evaluations or actions.

All these characteristics make the Zabbix proxy a simple, lightweight tool to deploy if you need to offload some checks from the central server or if your objective is to control and streamline the flow of monitored data across networks (possibly segregated by one or more firewalls) or both.

A basic distributed architecture involving Zabbix proxies would look as follows:



By its very nature, a Zabbix proxy should run on a dedicated machine, possibly different than the main server. A proxy is all about gathering data; it doesn't feature a frontend and it doesn't perform any complex queries or calculations; therefore, it's not necessary to assign a powerful machine with a lot of CPU power or disk throughput. In fact, a small, lean hardware configuration is often a better choice; proxy machines should be lightweight enough, not only to mirror the simplicity of the software component but also because they should be an easy and affordable way to expand and distribute your monitoring architecture without creating too much impact on deployment and management costs. A possible exception to the "small, lean, and simple" guideline for proxies can arise if you end up assigning hundreds of hosts with thousands of monitored items to a single proxy. In that case, instead of upgrading the hardware to a more powerful machine, it's often cheaper to just split up the hosts in different groups and assign them to different smaller proxies. In most cases, this would be the preferred option as you are not just distributing and evening out the load but you are also considering the possibility of huge data loss in case a single machine charged with the monitoring of a large portion of your network goes down for any reason. Consider using small, lightweight embedded machines as Zabbix proxies. They tend to be cheap, easy to deploy, reliable, and quite frugal when it comes to power requirements. These are ideal characteristics for any monitoring solution that aims to leave as little footprint as possible on the monitored system.

Deploying a Zabbix proxy

A Zabbix proxy is compiled together with the main server if you add `--enable-proxy` to the compilation options. The proxy can use any kind of database backend, just like the server, but if you don't specify an existing DB, it will automatically create a local SQLite to store its data. If you do intend to rely on SQLite, just remember to add `--with-sqlite3` to the options as well.

When it comes to proxies, it's usually advisable to keep things light and simple. A proxy DB will just contain some configuration and measurements data that, under normal circumstances, is almost immediately synchronized with the main server. Dedicating a full-blown database to it is usually overkill, so unless you have some very specific requirements, the SQLite option will provide the best balance between performance and ease of management.

If you didn't compile the proxy executable the first time you deployed Zabbix, just run `configure` again with the options you need for the proxies:

```
$ ./configure --enable-proxy --enable-static --with-sqlite3 --with-net-snmp --with-libcurl --with-ssh2 --with-openipmi
```

Compile everything again using the following command:

```
$ make
```

Beware that this will compile the main server as well; just remember not to run `make install`, or copy the new Zabbix server executable over the old one in the destination directory.

The only files you need to take and copy over to the proxy machine are the proxy executable and its configuration file. The `$PREFIX` variable should resolve to the same path you used in the configuration command (`/usr/local` by default):

```
# cp src/zabbix_proxy/zabbix_proxy $PREFIX/sbin/zabbix_proxy
# cp conf/zabbix_proxy.conf $PREFIX/etc/zabbix_proxy.conf
```

Next, you need to fill out relevant information in the proxy's configuration file. The default values should be fine in most cases, but you definitely need to make sure that the following options reflect your requirements and network status:

```
ProxyMode=0
```

This means that the proxy machine is in an active mode. Remember that you need at least as many Zabbix trappers on the main server as the number of proxies you deploy. Set the value to 1 if you need or prefer a proxy in the passive mode. See the *Understanding the flow of monitoring data with proxies* section for a more detailed discussion on proxy modes.

```
Server=n.n.n.n
```

This should be the IP number of the main Zabbix server or of the Zabbix node that this proxy should report to.

```
Hostname=Zabbix proxy
```

This must be a unique, case-sensitive name that will be used in the main Zabbix server's configuration to refer to the proxy.

```
LogFile=/tmp/zabbix_proxy.log
LogFileSize=1
DebugLevel=2
```

If you are using a small embedded machine, you may not have much disk space to spare. In that case, you may want to comment all the options regarding the logfile and let syslog send the proxy's log to another server on the Internet.

```
# DBHost=
# DBName=
# DBSchema=
# DBUser=
```

```
# DBPassword=
# DBSocket=
# DBPort=
```

Just leave everything commented out and the proxy will automatically create and use a local SQLite database. Fill out the relevant information if you are using a dedicated, external DB.

```
ProxyOfflineBuffer=1
```

This is the number of hours a proxy will keep monitored measurements if communications with the Zabbix server go down. You may want to double or triple it if you know you have a faulty, unreliable link between the proxy and server.

```
CacheSize=8M
```

This is the size for the configuration cache. Make it bigger if you have a large number of hosts and items to monitor.

The only thing that you need to do is to make the proxy known to the server, and add monitoring objects to it. All these tasks are performed through the Zabbix frontend.

The screenshot shows the Zabbix Administration web interface. The top navigation bar includes links for Monitoring, Inventory, Reports, Configuration, and Administration. The 'Administration' tab is selected. Below the navigation bar, there is a search bar and a 'ZBX main' link. The main content area is titled 'CONFIGURATION OF PROXIES' and shows the 'Proxy' configuration form. The form includes fields for 'Proxy name' (ZBX Proxy 1), 'Proxy mode' (Active), and two lists of hosts: 'Proxy hosts' (Delta, Gamma) and 'Other hosts' (Alpha, Beta, ZBX Main). At the bottom of the form are 'Save' and 'Cancel' buttons. The footer of the page displays 'Zabbix 2.0.6 Copyright 2001-2013 by Zabbix SIA' and 'Connected as 'Admin''.

Note how in the case of an **Active** proxy, you just need to specify the proxy's name, as already set in `zabbix_proxy.conf`. It will be the proxy's job to contact the main server. On the other hand, a **Passive** proxy will need an IP address or a hostname for the main server to connect to. See the *Understanding the flow of monitoring data with proxies* section for more details.

The screenshot shows the Zabbix Administration interface. The top navigation bar includes links for Monitoring, Inventory, Reports, Configuration, and Administration. The 'Administration' tab is selected. The breadcrumb trail indicates the path: Dashboard » Configuration of proxies » Configuration of host groups » Configuration of hosts » Configuration of proxies. The main heading is 'CONFIGURATION OF PROXIES'. Below this, the 'Proxy' configuration form is shown. The 'Proxy name' field contains 'ZBX Proxy 2'. The 'Proxy mode' dropdown is set to 'Passive'. The 'Interface' section is highlighted with a dashed border and contains four sub-fields: 'IP address' (10.2.9.11), 'DNS name' (zbxprx2.example.net), 'Connect to' (with 'IP' and 'DNS' buttons), and 'Port' (10051). Below the interface fields, there are two list boxes for 'Hosts'. The 'Proxy hosts' list contains 'Alpha'. The 'Other hosts' list contains 'Beta', 'Delta', 'Gamma', and 'ZBX Main'. At the bottom of the form are 'Save' and 'Cancel' buttons. The footer of the page shows 'Zabbix 2.0.6 Copyright 2001-2013 by Zabbix SIA' and 'Connected as 'Admin''.

You can assign hosts to proxies during the proxy creation process, or in the proxy's edit screen, or even from the host's configuration screen, as in the following screenshot:

The screenshot displays the Zabbix web interface for configuring a host. The top navigation bar includes tabs for Configuration, Administration, and others. The breadcrumb trail indicates the path: Configuration of proxies » Configuration of host groups » Configuration of hosts » Configuration of proxies » Configuration of hosts. The main title is 'CONFIGURATION OF HOSTS'. Below this, there are links for 'list', 'Host: Beta', 'Monitored', and various metrics like 'Applications (10)', 'Items (32)', 'Triggers (15)', 'Graphs (4)', and 'Discovered hosts'. The 'Host inventory' tab is selected, showing a list of host groups. The 'In groups' section lists 'Linux servers'. The 'Other groups' section lists 'Discovered hosts', 'Templates', 'Windows servers', and 'Zabbix servers'. Below this, there is a 'New host group' section with a text input field. The 'Agent interfaces' section shows a table with columns for IP address, DNS name, and Connection type. The 'IP address' column contains '127.0.0.1'. The 'DNS name' column is empty. The 'Connection type' column has a button labeled 'IP'. Below the table, there are 'Add' buttons for 'Agent interfaces', 'SNMP interfaces', 'JMX interfaces', and 'IPMI interfaces'. The 'Monitored by proxy' dropdown menu is open, showing options: '(no proxy)', '(no proxy)', 'ZBX Proxy 1', and 'ZBX Proxy 2'. The 'Status' dropdown menu is also open, showing options: '(no proxy)', 'ZBX Proxy 1', and 'ZBX Proxy 2'. At the bottom, there are buttons for 'Save', 'Clone', 'Full clone', 'Delete', and 'Cancel'.

Agent interfaces	IP address	DNS name	Connection type
	127.0.0.1		IP

Monitored by proxy: (no proxy) (no proxy) ZBX Proxy 1 ZBX Proxy 2

Status: (no proxy) ZBX Proxy 1 ZBX Proxy 2

Buttons: Save Clone Full clone Delete Cancel

One of the advantages of proxies is that they don't need much configuration or maintenance; once they are deployed and you have assigned some hosts to one of them, the rest of the monitoring activities are fairly transparent. Just remember to check the number of values per second every proxy has to guarantee as expressed by the **Required performance** column in the proxies' list page:

ZABBIX Help | Get support | Print | Profile | Logout

Monitoring | Inventory | Reports | Configuration | Administration ZBX main

General | **DM** | Authentication | Users | Media types | Scripts | Audit | Queue | Notifications | Installation Search

History: Configuration of proxies » Configuration of host groups » Configuration of hosts » Configuration of proxies » Configuration of hosts

CONFIGURATION OF PROXIES Proxies Create proxy

Proxies

Displaying 1 to 2 of 2 found

<input type="checkbox"/>	Name	Mode	Last seen (age)	Host count	Item count	Required performance (vps)	Hosts
<input type="checkbox"/>	ZBX Proxy 1	Active	27s	2	36	0.6	Gamma , Delta
<input type="checkbox"/>	ZBX Proxy 2	Passive	-	1	32	0.46	Alpha

Enable selected Go (0)

Zabbix 2.0.6 Copyright 2001-2013 by Zabbix SIA | Connected as 'Admin'

Values per second (vps) is the number of measurements per seconds that a single Zabbix server or proxy has to collect. It's an average value that depends on the number of items and the polling frequency for every item. The higher the value, the more powerful the Zabbix machine must be.

Depending on your hardware configuration, you may need to redistribute the hosts among proxies or add new ones in case you notice degraded performances coupled with high vps.

Understanding the flow of monitoring data with proxies

Zabbix proxies can operate in two different modes, active and passive. An active proxy, which is the default setup, initiates all connections to the Zabbix server, both to retrieve configuration information on monitored objects and to send measurements back to be further processed. You can tweak the frequency of these two activities by setting the following variables in the proxy configuration file:

```
ConfigFrequency=3600
```

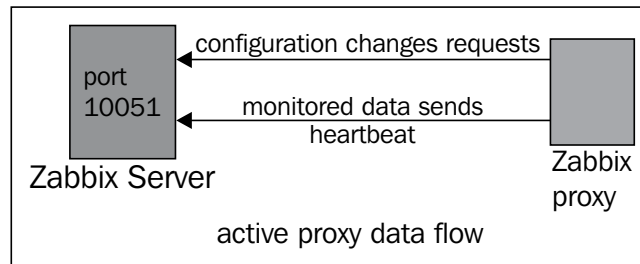
```
DataSenderFrequency=1
```

Both the preceding values are in seconds. On the server side, in the `zabbix_server.conf` file, you also need to set the value of `StartTrappers=` to be higher than the number of all active proxies and nodes you have deployed. The trapper processes will have to manage all incoming information from proxies, nodes, and any item configured as an active check. The server will fork extra processes as needed, but it's advisable to prefork as many processes that you already know the server will use.

Back on the proxy side, you can also set a `HeartbeatFrequency` so that after a predetermined number of seconds, it will contact the server even if it doesn't have any data to send. You can then check on the proxy availability with the following item, where `proxy name` of course is the unique identifier that you assigned to the proxy during deployment:

```
zabbix[proxy, "proxy name", lastaccess]
```

The item as expressed will give you the number of seconds since the last contact with the proxy, a value you can then use with the appropriate triggering functions. A good starting point to finetune the optimal heartbeat frequency is to evaluate how long you can afford to lose contact with the proxy before being alerted, and consider that the interval is just over two heartbeats. For example, if you need to know if a proxy is possibly down in less than five minutes, set the heartbeat frequency to 120 seconds and check whether the last access time was above 300 seconds.



An active proxy is more efficient in offloading computing duties from the server as the latter will just sit idle, waiting to be asked about changes in configuration or to receive new monitoring data. The downside is that proxies will often be deployed to monitor secure networks, such as DMZs and other segments with strict outgoing traffic policies. In these scenarios, it would be very difficult to obtain permission for the proxy to initiate contact with the server. And it's not just a matter of policies; DMZs are isolated as much as possible from internal networks for extremely good and valid reasons. On the other hand, it's often easier and more acceptable from a security point of view to initiate a connection from the internal network to a DMZ. In these cases, a passive proxy will be the preferred solution.

Connection- and configuration-wise, a passive proxy is almost the mirror image of the active version. This time, it's the server that needs to connect periodically to the proxy to send over configuration changes and to request any measurements the proxy may have taken. On the proxy configuration file, once you've set `ProxyMode=1` to signify that this is a passive proxy, you don't need to do anything else. On the server side, there are three variables you need to check:

- `StartProxyPollers=`

This represents the number of processes dedicated to manage passive proxies and should match the number of passive proxies you have deployed.

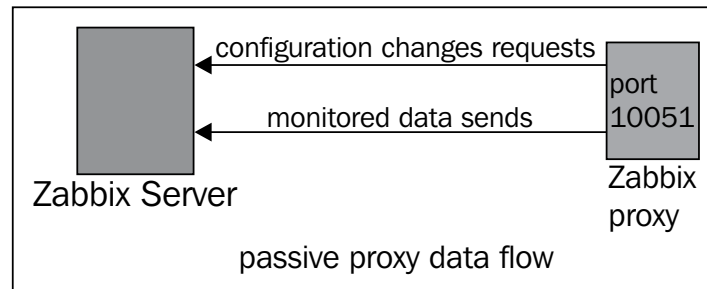
- ProxyConfigFrequency=

The server will update a passive proxy with configuration changes for the number of seconds you have set in the preceding variable.

- ProxyDataFrequency=

This is the interval, also in seconds, between two consecutive requests by the server for the passive proxy's monitoring measurements.

There are no further differences between the two modes of operation for proxies. You can still use the `zabbix[proxy, "proxy name", lastaccess]` item to check a passive proxy's availability, just like for the active one.

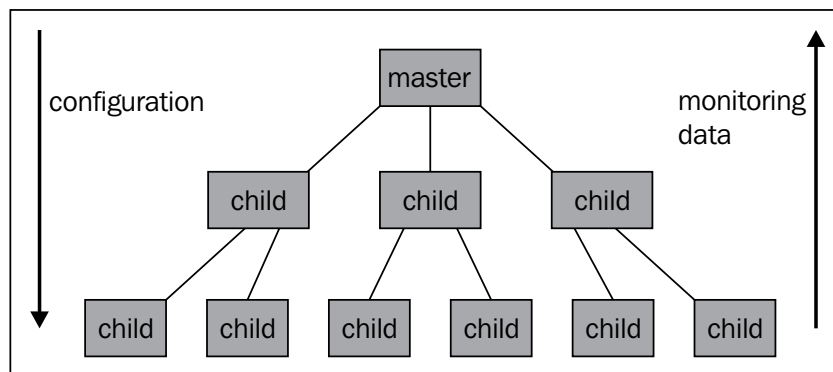


At the price of a slightly increased workload for the server, when compared to active proxies, a passive one will enable you to gather monitoring data from otherwise closed and locked-down networks. At any rate, you can mix and match active and passive proxies in your environment, depending upon the flow requirements of specific networks. This way, you will significantly expand your monitoring solution both in its ability to reach every part of the network and in its ability to handle a large number of monitored objects, while at the same time keeping the architecture simple and easy to manage with a strong central core and many simple, lightweight yet effective satellites.

Zabbix nodes

While proxies are often an adequate solution to many distributed monitoring problems, sometimes they just lack the features needed in some scenarios. Consider, for example, the case where you not only have a huge number of items but also a great number of complex and computing-intensive triggers that can really impact your server performance. Proxies won't be able to tackle the heart of the problem in this case. Another case would be that of a group merger of different companies with an IT infrastructure that's brought together but still managed by different teams, which still need a bit of operational independence within a shared context. Or you could have some of your company's branches in distant geographical locations and it just doesn't make sense to occupy a significant portion of your bandwidth with every single detail of monitored data. All these scenarios and others can be managed, thanks to the Zabbix server's ability to take part in a larger tree-like structure of monitoring nodes.

A Zabbix node is a full-blown server with its own database, frontend, authentication sources, and monitored objects that has been assigned a node ID and has been given a place in the hierarchy of Zabbix nodes. As part of this hierarchy, a Zabbix node will get configuration information from the parent nodes and report back to them on monitoring measurements and trigger events, while distributing configuration information to child nodes and retrieve monitoring data from them. A node tree is strictly hierarchical, and nodes on the same level (that is, sharing the same parent) won't exchange data in any way.



Understanding the flow of data with nodes

A parent node will be able to manage almost any aspect of all of its child nodes' configuration, at any level down the tree. This is not a real-time operation; every parent keeps a copy of all its child nodes' configuration data (hosts, items, triggers, templates, screens, and so on) in the database, and periodically calculates a checksum to compare with each child node. If there are any differences, the child updates its configuration with the new information sent from the parent.

Conversely, a child can update the information a parent has about it using the same mechanism. A checksum is sent over to the parent. If there are any differences, the parent updates its information about the child with the new data sent from the child itself. This works with modifications to a child node's configuration done directly from a parent node or with updates from the child to the parent only.

Due to current limitations in the nodes' architecture, there's no easy way to move or copy configuration information (templates, for example) from one node to another using the checksum feature, not even from the parent to the child. If you need to create new templates and distribute them to all nodes, you will have to resort to some out-of-band strategy. We will see a of possible solution in the following chapters.

Deploying a node

A Zabbix node is first and foremost a regular Zabbix server installation. You can refer to *Chapter 1, Deploying Zabbix*, for indications on server- and database-choice and deployment.

Once you have installed a Zabbix server, there are a few operations you need to perform in order to make it into a node that is part of a specific hierarchy. You will need to make a few changes to the server configuration file, to the database, and to both the parent's and child's frontends.

First of all, you need to assign a server a node ID number. Zabbix supports up to 1000 nodes, so any number between 0 and 999 will do. You don't need to respect a sequential order for the nodes hierarchy. A node ID is just a label and it doesn't matter if Node 1 is a child of Node 57 or vice versa. On the other hand, in order for every monitored object to be unique across the whole distributed node structure, this number will be prepended to an object ID in the database of a node. These IDs will then be synchronized with the parent nodes, as explained in the previous section, so be sure that every node ID will be unique in your nodes' hierarchy.

A database table for a Zabbix installation that is not part of a node structure will look like this:

```
> select itemid, name from items limit 5;
+-----+-----+
| itemid | name                                     |
+-----+-----+
| 10009  | Number of processes                     |
| 10010  | Processor load (1 min average per core) |
| 10013  | Number of running processes            |
| 10014  | Free swap space                         |
| 10016  | Number of logged in users               |
+-----+-----+
```

Note the range of the item IDs. The same ranges can be found for host IDs and any other monitoring object such as triggers, graphs, maps, and map items.

The main operation to turn a Zabbix installation into a node is to perform an update of the database, by issuing the following command with the node ID you have assigned to the server.

```
$ zabbix_server -n <node id>
```

For example, assuming that you have assigned node ID 99 to your Zabbix server, you would issue the following command:

```
$ zabbix_server -n 99
Dropping foreign keys ..... done.
Converting tables ..... done.
Creating foreign keys ..... done.
Conversion completed successfully.
```

This will change all the existing and future object IDs by adding the node ID to them:

```
> select itemid, name from items limit 5;
+-----+-----+
| itemid          | name                                     |
+-----+-----+
| 9909900000010009 | Number of processes                     |
| 9909900000010010 | Processor load (1 min average per core) |
| 9909900000010013 | Number of running processes            |
| 9909900000010014 | Free swap space                         |
| 9909900000010016 | Number of logged in users               |
+-----+-----+
```

As you can see, the same item IDs have now been prepended with the node ID you have chosen. Before logging on the frontend, you still have to specify the same node ID in the server configuration file. Once done, just restart the server daemons and you'll immediately see that the frontend will reflect the recent changes. You'll be able to select a node from a drop-down menu and all objects will report what node they are part of. By going to **Administration** | **DM** | **Nodes**, you can change the name of the local node to reflect the name of the server, so that it's never ambiguous as to what "local node" you might refer to in your interface:

ZABBIX Help | Get support | Print | Profile | Logout

Monitoring | Inventory | Reports | Configuration | **Administration** | ZBX main | Current node: ZBX Main ▼ | Select Nodes

General | **DM** | Authentication | Users | Media types | Scripts | Audit | Queue | Notifications | Installation

History: Dashboard » Overview » Latest data » Configuration of proxies » Configuration of nodes

CONFIGURATION OF NODES

Node

Name: ZBX Main

ID: 99

Type: Local

IP: 10.2.2.1

Port: 10051

Save **Cancel**

Zabbix 2.0.6 Copyright 2001-2013 by Zabbix SIA | Connected as 'Admin' from 'ZBX Main'

At this point, you just need to repeat the same procedure for all the nodes in your hierarchy, and you'll be ready to link them in a tree-like structure. Double-check your node IDs! You can only execute the `zabbix_server -n <node id>` command once. Any other attempt on the same database will corrupt it. So make sure that you are not re-using any node ID or you'll have to start from scratch for the node you "cloned".

For every parent-child link, you'll need to configure the relationship on both nodes. Starting from the parent, make sure that you select the right node ID, make sure the type is "child", and whether you have selected the right parent (**Master node**). Keep in mind that parent nodes are identified by their node names and not by their IDs in the interface, so be sure to keep your nodes names and ID's map handy.

ZABBIX Help | Get support | Print | Profile | Logout

Monitoring | Inventory | Reports | Configuration | **Administration** | ZBX main | Current node: ZBX Main ▼ | Select Nodes

General | **DM** | Authentication | Users | Media types | Scripts | Audit | Search

Queue | Notifications | Installation

History: Configuration of proxies » Configuration of nodes » Dashboard » Configuration of proxies » Configuration of nodes

CONFIGURATION OF NODES

Node

Name: ZBX Sat 1

ID: 77

Type: Child ▼

Master node: ZBX Main ▼

IP: 10.7.9.1

Port: 10051

Save Cancel

Zabbix 2.0.6 Copyright 2001-2013 by Zabbix SIA | Connected as 'Admin' from 'ZBX Main'

On the child's side, create a node with the same node ID, its name as the parent, and specify **Type** as **Master**, as shown in the following screenshot:

ZABBIX Help | Get support | Print | Profile | Logout

Monitoring | Inventory | Reports | Configuration | **Administration** | Current node: ZBX Sat 1 | Select Nodes

General | **DM** | Authentication | Users | Media types | Scripts | Audit | Queue | Notifications | Installation

History: Dashboard » Configuration of proxies » Configuration of nodes

CONFIGURATION OF NODES

Node

Name: ZBX Main

ID: 99

Type: Master ▼

IP: 10.2.9.11

Port: 10051

Zabbix 2.0.6 Copyright 2001-2013 by Zabbix SIA | Connected as 'Admin' from 'ZBX Sat 1'

The two nodes are now linked. You will be able to completely administer the child node from the parent one, and the two nodes will start to synchronize both configuration and monitoring data as explained in the previous section.

You can control what kind of data a child sends to its master node by setting the variables:

```
NodeNoEvents=
NodeNoHistory=
```

Setting both the values to 1 will instruct the child node not to send the event information or history data, respectively. This won't have any impact on the data sent from nodes further down the branch. A node will always pass over to its master any data received from its child nodes. The configuration update and data-send frequency are fixed for nodes: two minutes for configuration and five seconds for history and events.

Once you have historical data or events from child nodes in your master node, you can use it to create further aggregate items, triggers, or graphs, just like any other monitoring object.

Proxies versus nodes

At first sight, it could seem that nodes are much more powerful and flexible than proxies. While that is certainly true to a certain extent, it doesn't mean that they are automatically the preferred solution to every distributed monitoring scenario. In fact, there are a number of drawbacks to using nodes.

Disadvantages of nodes

First of all, every node is a full server, with a full-blown database and web frontend. This means more expensive hardware than that required by a proxy, but most of all, this means a lot more maintenance in terms of day-to-day system and database administration, backups, security policy compliance, and so on. It's also not easy to move nodes around once they are configured, and while they do send data upstream to the master node, they still need to be managed one-by-one as single entities. You can't mass-update hosts or items across nodes, for example. More importantly, if you update a template that is used by more than one node, you'll have to replicate the update to the other nodes individually. You can try to automate template synchronization by leveraging the power of the Zabbix API and a lot of custom code, and in *Chapter 9, Extending Zabbix*, we'll show you an example of such a setup, but you'll have to enforce strong template-update policies throughout your node tree, or you could find yourself with inconsistent data.

Speaking of access policies, these too have to be managed node-by-node, and for every child node, you'll have to decide not only the group permissions on the node itself, like a standalone installation, but you'll also have to go up the branch and figure out group permissions for that node's data on every parent node.

Finally, since configuration-synchronization is performed by calculating checksums, this operation can add up to a significant amount of CPU time for a master node, such that you may need to invest in more powerful hardware just to keep up with synchronization. A master node's database size tends to grow significantly too as it has to keep information about the child nodes as well.

By contrast, none of these drawbacks exist with proxies. They may be simpler and less powerful, but they are also far easier to deploy and maintain and easier to move around in case you need to, and they also consume less computing resources and less disk space overall.

Choosing between proxies and nodes

To summarize, the power of a node's base architecture comes at a cost:

- It requires more powerful hardware
- It is more complex to configure
- It requires heavy maintenance
- It is not very flexible

On the other hand, a proxy-based architecture is:

- Easy to deploy
- Requires simple hardware
- Easy to configure
- More flexible than a hierarchy of nodes

All in all, a reasonable piece of advice would be to go with proxies whenever possible and resort to nodes when necessary. Cases such as the merging of two distinct networks with existing Zabbix installations, distant geographical branches of the same network or company, and really huge networks where every node has to manage thousands of nodes are good examples of scenarios where a node-based solution works best. A typical large environment will probably have a fair amount of proxies but only a limited number of nodes, such that any maintenance overhead would be contained as much as possible.

Security considerations

One of the few drawbacks of the whole Zabbix architecture is the lack of built-in security at the Zabbix protocol level. While it's possible to protect both the web frontend and the Zabbix API by means of a standard SSL layer to encrypt communications, and relying on different authorities for identification, there's simply no standard way to protect communications between agents and server, or between proxies and server, or among nodes, not even when it comes to message authentication (the other party is indeed who it says it is), nor when it comes to message integrity (the data has not been tampered with) and neither when it comes to message confidentiality (no one else can read or understand the data).

If you've been paying attention to the configuration details of agents, proxies, and nodes, you may have noticed that all that a Zabbix component needs to know in order to communicate to another component is its IP address. No authentication is performed as relying on only the IP address to identify a remote source is inherently insecure. Moreover, any data sent is clear text, as you can easily verify by running `tcpdump` (or any other packet sniffer):

```
$ zabbix_sender -v -z 10.10.2.9 -s alpha -k sniff.me -o "clear text data"
```

```
$ tcpdump -s0 -nn -q -A port 10051
00:58:39.263666 IP 10.10.2.11.43654 > 10.10.2.9.10051: tcp 113
E....l@.P.....'C...' ^.....V.....
.Gp|.Gp|{
    "request":"sender data",
    "data":[
        {
            "host":"alpha",
            "key":"sniff.me",
            "value":"clear text data"}}}]}
```

Sure, simple monitoring or configuration data may not seem much, but at the very least, if tampered with, it could lead to false and unreliable monitoring.

While there are no standard counter measures to this problem, there are a few possible solutions to it that increase in complexity and effectiveness from elementary, but not really secure to complex and reasonably secure. Keep in mind that this is not a book on network security, so you won't find any deep, step-by-step instructions on how to choose and implement your own VPN solution. What you will find is a brief overview of methods to secure the communication between the Zabbix components, which will give you a practical understanding of the problem, so you can make an informed decision on how to secure your own environment.

No network configuration

If, for any reason, you can't absolutely do anything else, you should at the very least specify a source IP for every Zabbix trapper item, so that it wouldn't be too easy and straightforward to spoof monitoring data using the `zabbix_sender` utility. Use the macro `{HOST.CONN}` in a template item so that every host will use its own IP address automatically:

Item

Host

Template OS Linux

Select

Name

Type

Zabbix trapper

Key

Select

Type of information

Numeric (unsigned)

Data type

Decimal

Units

Use custom multiplier

☐

1

Keep history (in days)

90

Keep trends (in days)

365

Store value

As is

Show value

As is

[show value mappings](#)

Allowed hosts

{HOST.CONN}

New application

Applications

-None-

CPU

Filesystems

General

Memory

Network interfaces

Populates host inventory field

-None-

Description

Status

Enabled

Save

Cancel

More importantly, make sure that remote commands are not allowed on agents. That is, `EnableRemoteCommands` in the `zabbix_agentd.conf` file must be set to 0. You may lose a convenient feature, but if you can't protect and authenticate the server-agent communications, the security risk is far too great to even consider taking it.

Network isolation

Many environments have a management network that is separated and isolated from your production network via nonrouted network addresses and VLANs. Network switches, routers, and firewalls typically handle traffic on the production network, but are reachable and can be managed only through their management network address. While this makes it a bit less convenient to access them from any workstation, it also makes sure that any security flaw in your components (consider, for example, a network appliance that has a faulty SSL implementation that you can't use or that doesn't support SNMP v3, or has Telnet inadvertently left open) is contained to a separated and difficult-to-reach network. You may want to put all of the server-proxy and master-child communications on such an isolated network. You are just making it harder to intercept monitoring data and you may be leaving out the server-agent communications, but isolating traffic is still a sensible solution even if you are going to further encrypt it with one of the solutions outlined in the following sections.

On the other hand, you certainly don't want to use this setup for a node or proxy that is situated in a DMZ or another segregated network. It's far more risky to bypass a firewall through a management network than to have your monitoring data pass through the said firewall. Of course, this doesn't apply if your management network is also routed and controlled by the firewall, but it's strongly advised to verify that this is indeed the case before looking into using it for your monitoring data.

Simple tunnels

So far, we haven't really taken any measures to secure and encrypt the actual data that Zabbix sends or receives. The simplest and most immediate way to do that is to create an ad hoc encrypted tunnel through which you can channel your traffic.

Secure Shell

Fortunately, **Secure Shell (SSH)** has built-in tunneling abilities, so if you have to encrypt your traffic in a pinch, you already have all the tools you need.

To encrypt the traffic from an active proxy to the server, just log on the proxy's console and issue a command similar to the following one:

```
$ ssh -N -f user@zabbix.server -L 10053:localhost:10051
```

In the preceding command, `-N` means that you don't want the SSH client to execute any commands, other than just routing the traffic; the `-f` option makes the SSH client go into the background (so you don't have to keep a terminal open, or a start script executing forever), `user@zabbix.server` is a valid user (and real hostname or IP address) on the Zabbix server, and the `-L port:remote-server:port` sets up the tunnel. The first port number is what your local applications will connect to, while the following `host:port` combination specifies what host and TCP port the SSH server should connect to as the other end of the tunnel.

Now set your `Server` and `ServerPort` options in your `zabbix_proxy.conf` to `localhost` and `10053` respectively.

What will happen is that from now on, the proxy will send data to port 10053 by itself, where there's an SSH tunnel session waiting to forward all traffic via the SSH protocol to the Zabbix server. From there, the SSH server will in turn forward it to a local port 10051 and finally to the Zabbix daemon. While all of the Zabbix components don't natively support data encryption for the Zabbix protocol, you'll still be able to make them communicate while keeping message integrity and confidentiality; all you will see on the network with such a setup will be standard, encrypted SSH traffic data on TCP port 22.

To make a Zabbix server contact a Passive proxy via a tunnel, just set up a listening SSH server on the proxy (you should already have it in order to remotely administrate the machine) and issue a similar command as the one given earlier on the Zabbix server, making sure to specify the IP address and a valid user for the Zabbix proxy. Change the proxy's IP address and connection-port specifications on the web frontend, and you are done.

To connect to Zabbix nodes, you need to set up two such tunnels, one from the master to the child and one from the child to the master.

On the master, run the following command:

```
$ ssh -N -f user@zabbix.child -L 10053:localhost:10051
```

On the child, run the following command:

```
$ ssh -N -f user@zabbix.master -L 10053:localhost:10051
```

Stunnel

Similar functionalities can be obtained using the stunnel program. The main advantage of using stunnel over SSH is that with stunnel, you have a convenient configuration file where you can set up and store all your tunneling configurations, while with SSH, you'll have to script the preceding commands somehow if you want the tunnels to be persistent across your machine's reboots.

Once installed, and once you have created the copies of the obtained SSL certificates that the program needs, you can simply set up all your port-forwarding in the `/etc/stunnel/stunnel.conf` file. Considering, for example, a simple scenario with a Zabbix server that receives data from an active proxy and exchanges data with another node, after having installed stunnel and SSL certificates on all three machines, you could have the following setup.

On the Zabbix server's `stunnel.conf`, add the following lines:

```
[proxy]
accept = 10055
connect = 10051

[node - send]
accept = localhost:10057
connect = node.server:10057

[node - receive]
accept = 10059
connect = 10051
```

On the Zabbix proxy's `stunnel.conf`, add the following lines:

```
[server]
accept = localhost:10055
connect = zabbix.server:10055
```

On the other node's `stunnel.conf`, add the following lines:

```
[node - send]
accept = localhost:10059
connect = node.server:10059

[node - receive]
accept = 10057
connect = 10051
```

Just remember to update the host and port information for proxies and servers in their respective configuration files and web frontend forms.

As you can see, the problem with port-forwarding tunnels is that the more tunnels you set up, the more different ports you have to specify. If you have a large number of proxies and nodes, or if you want to encrypt the agent data as well, all the port forwarding will quickly become cumbersome to set up and keep track of. This is a good solution if you just want to encrypt your data on an insecure channel among a handful of hosts, but if you want to make sure that all your monitoring traffic is kept confidential, you'll need to resort to a more complete VPN implementation.

A full-blown VPN

This is not the place to discuss the relative merits of different VPN implementations, but if you do use a VPN solution in your network, consider switching all Zabbix monitoring to your encrypted channel. Of course, unless you want the whole world to look at your monitoring data, this is practically mandatory when you link two nodes or a server and a proxy, from distant geographical locations that are connected only through the Internet. In that case, you hopefully already have a VPN, whether a simple SSL one or a full-blown IPSEC solution. If you don't have it, protecting your Zabbix traffic is an excellent reason to set up one.

These workarounds will protect your traffic, and in the best-case scenario, will provide basic host authentication, but keep in mind that until Zabbix supports some sort of security protocol on the application level, tunneling and encryption will only be able to protect the integrity of your monitoring data. Any user who gains access to a Zabbix component (whether it's a server, proxy, or agent) will be able to send bogus data over the encrypted channel, and you'll have no way to suspect foul play. So, in addition to securing all communication channels, you also need to make sure that you have good security at the host level.

Summary

In this chapter, we saw how to expand a simple, standalone Zabbix installation into a vast and complex distributed monitoring solution. By now, you should be able to understand how Zabbix proxies and nodes work, how they pass monitoring information around, what their respective strong points and possible drawbacks are, and what is their impact in terms of hardware requirements and maintenance.

You should also know when and how to choose between an active proxy and a passive one, when to switch to a node-based implementation, and more importantly, how to mix and match the two features into a tailor-made solution for your own environment.

Finally, you should have a clear understanding of how to evaluate possible security concerns regarding monitored data and what possible measures you can take to mitigate security risks related to a Zabbix installation.

In the next chapter, we will conclude with an overview on how to deploy Zabbix in a large environment by talking about high availability at the three levels of database, monitoring server, and web frontend.

3

High Availability and Failover

Now that you have a good knowledge of all the components of a Zabbix infrastructure, it is time to implement a highly available Zabbix installation. In a large environment, especially if you need to guarantee that all your servers are up and running, it is of vital importance to have a reliable Zabbix infrastructure. The monitoring system and Zabbix infrastructure should survive any possible disaster and guarantee business continuity.

High availability is one of the solutions that guarantees business continuity and provides a disaster recovery implementation; this kind of setup cannot be missed in this book.

This chapter begins with the definition of high availability, and it further describes how to implement an HA solution.

In particular, this chapter considers the three-tier setup that we described earlier:

- The Zabbix GUI
- The Zabbix server
- Database

For each one of those components, we have described how to set up and configure it on high availability. All the procedures presented in this chapter have been implemented and tested on a real environment.

In this chapter, we will cover the following topics:

- Understanding what is high availability, failover, and its service level
- In-depth analysis of all the components (the Zabbix server, the web server, and the RDBMS server) of our infrastructure and how they will fit in a highly available installation
- Implement a highly available setup of our monitoring infrastructure

Understanding high availability

High availability is an architectural design approach and associated service implementation that is used to guarantee the reliability of a service. Availability is directly associated to the uptime and usability of a service. This means that the downtime should be reduced to achieve an agreement on that service.

We can distinguish between two kinds of downtime:

- Scheduled or planned downtime
- Unscheduled or unexpected downtime

To distinguish between the scheduled downtime, we can include:

- System patching
- Hardware expansion or hardware replacement
- Software maintenance
- All that is normally a planned maintenance task

Unfortunately, all these downtimes will interrupt your service, but you have to agree that they can be planned into a maintenance window agreed.

The unexpected downtime normally arises from a failure, and it can be caused by one of the following reasons:


- Hardware failure
- Software failure
- Physical events

Unscheduled downtimes also include the power outage and over-temperature shutdown, and all these are not planned; however, they cause an outage. Hardware and software failure are quite easy to understand, whereas a physical event is an external event that produces an outage on our infrastructure. A practical example can be an outage that can be caused by lightning or a flood that leads to the breakdown of the electrical line with consequences on our infrastructure. The availability of a service is considered from the service user's point of view; for example, if we are monitoring a web application, we need to consider this application from the web user's point of view. This means if all your servers are up and running, but a firewall is cutting connections and the service is not accessible, this service cannot be considered available.


Understanding the levels of IT service

Availability is directly tied with service level and is normally defined as a percentage. It is the percentage of uptime over a defined period. The availability that you can guarantee is your service level. The following table shows what exactly this means by considering the maximum admitted downtime for some of the frequently used availability percentage.


Availability percentage	Max downtime per year	Max downtime per month	Max downtime per week
90% called "one nine"	36.5 days	72 hours	16.8 hours
95%	18.25 days	36 hours	8.4 hours
99% called "two nines"	3.65 days	7.20 hours	1.68 hours
99.5%	1.83 days	3.60 hours	50.4 minutes
99.9% called "three nines"	8.76 hours	43.8 minutes	10.1 minutes
99.95%	4.38 hours	21.56 minutes	5.04 minutes
99.99% called "four nines"	52.56 minutes	4.32 minutes	1.01 minutes
99.999% called "five nines"	5.26 minutes	25.9 seconds	6.05 seconds
99.9999% called "six nines"	31.5 seconds	2.59 seconds	0.605 seconds
99.99999% called "seven nines"	3.15 seconds	0.259 seconds	0.0605 seconds

 Uptime is not a synonym of availability. A system can be up and running but not available; for instance, if you have a network fault, the service will not be available, but all the systems will be up and running.


The availability must be calculated end-to-end, and all the components required to run the service must be available. The next sentence may seem a paradox; the more hardware you add and the more failure points you need to consider, the more it becomes increasingly difficult to implement an efficient solution. Also, an important point to consider is how easy the patching of your HA system and its maintenance will be. A true highly available system implies that human intervention is not needed; for example, if you need to agree to a five nine service level, the human (your system administrator) will have only one second of downtime per day, so here the system must respond to the issue automatically. Instead, if you agree to a two nines **service level agreement (SLA)**, the downtime per day can be 15 minutes; here, the human intervention is realistic but unfortunately this SLA is not a common case. Now while agreeing to an SLA, the mean time for recovery is an important factor to consider.

 **Mean Time To Recovery (MTTR)** is the mean time that a device will take to recover from a failure.

The first thing to do is to keep the architecture as simple as possible and reduce the number of actors in the play to minimum. Simpler the architecture, lesser the effort required to maintain, administer, and monitor it. All that the HA architecture needs is to avoid a single point of failure, and it needs to be as simple as possible. For this reason, the solutions here presented are easy to understand, tested on production environments, and quite easy to implement and maintain.

[ Complexity is the first enemy of high availability.]

Unfortunately, a highly available infrastructure is not designed to achieve the highest performance possible. This is because it is normal that an overhead will be introduced to keep two servers updated and a highly available infrastructure is not designed for the maximum throughput. Also, if there are implementations that consider using the standby server as a read-only server to reduce the load of a primary node, use an unused server.

[ A highly available infrastructure is not designed to achieve the maximum performance or throughput.]

Some consideration about high availability

Every HA architecture has some common problem to solve or common question to respond:

- How to handle the connection?
- How to manage the failover?
- How the storage is shared or replicated to the other site?

There are some production-stable and widely used solutions for each one of those questions. Let us study these questions in detail:

- How to handle the connection?

One of the possible responses to this question is one word – **VIP (Virtual IP)**. Basically, every software component needs to communicate or is interconnected with different logical layers, and those components are often deployed on different servers to divide and equalize the workload. Most of the communications are TCP/IP based, and here the network protocol gives us a hand.

It is possible to define a VIP that is assigned to the active servers and all the software required to be configured to use that address. So in case of a failover, the IP address will follow the service and all the clients will continue to work. Of course, this solution can't guarantee that there isn't downtime at all, but it will be time limited and for a small period. From the administration point, apart from checking the failover, the administrator doesn't need to reconfigure anything.

- How to manage the failover?

The response to this question is: use a resource manager. You need to think of a smart way to move a faulty service to the standby node as soon as possible that is independent of SLA. To achieve the minimum downtime possible, you need to automate the service failover on the standby node, and to give the business continuity, the fault needs to be found as soon as possible when it happens.

- How the storage is shared or replicated to the other site?

This last question can be implemented with different actors, technologies, and methodologies. You can use a shared disk, a replicated **Logical Unit Number (LUN)** between two storages, or use a replicated device with software. Unfortunately, using a replicated LUN between two storages is quite expensive. This software should be more close to kernel and should be working on the lowest layer possible to be transparent from the operating system's perspective, thereby keeping things easy to manage.

Automating the switchover/the failover with a resource manager

The architecture that you are going to implement needs a component to automate the switchover or the failover, basically, as said earlier, it requires a resource manager.

One of the resource managers that is widely used and is production mature is Pacemaker. Pacemaker is an open source high availability resource manager designed for small and large clusters. Pacemaker is available for download at the following URL: <http://clusterlabs.org/>

Pacemakers provide some interesting features that are really useful for your cluster:

- Detecting and recovering server issues at the application level
- Supporting redundant configurations
- Supporting multiple nodes applications
- Supporting startup/shutdown ordering applications

Practically, a Pacemaker replaces you and is automated and fast. Pacemaker does the work that a Unix administrator normally does in case of node failure. It checks if the service is no more available and switches all the configured services on the spare node; plus, it does all this work as quickly as possible. This switchover gives us the time required to do all the forensic analysis while all the services are still available. In another context, the service would be simply unavailable.

There are different solutions that provide cluster managements. Red Hat Cluster Suite is a popular alternative. It is not proposed here as it is not really completely tied to Red Hat, however, it is definitely developed with this distribution in mind.

Replicating the filesystem with DRBD

Distributed Replicated Block Device (DRBD) has some features that are point of force for this solution:

- It is a kernel module
- It is completely transparent from the point of view of RDBMS
- It provides real-time synchronization
- It synchronizes writes on both nodes
- It automatically performs resynchronization
- It practically acts like a networked RAID-1

The core functionality of DRBD is implemented on the kernel layer, in particular DRBD is a driver for a virtual block device, so DRBD works at the bottom of the system I/O stack.

DRBD can be considered equivalent to a networked RAID-1, below the OS's filesystem, at the block level.

This means that DRBD synchronization is synced to the filesystem. The worst scenario and more complex to handle is a filesystem replication for an RDBMS. In this case, every commit needs to be acknowledged on both nodes before it happens, and all the committed transaction is written on both nodes; DRBD completely supports this case.

Now what happens when a node is no longer available? It's simple; DRBD will operate exactly like a degraded RAID-1. This is a strong point because if your Disaster Recovery site goes down, you don't need to do anything. Once the node reappears, DRBD will do all the synchronization work for us, that is, rebuilding or resynchronizing the offline node.

Implementing high availability on a web server

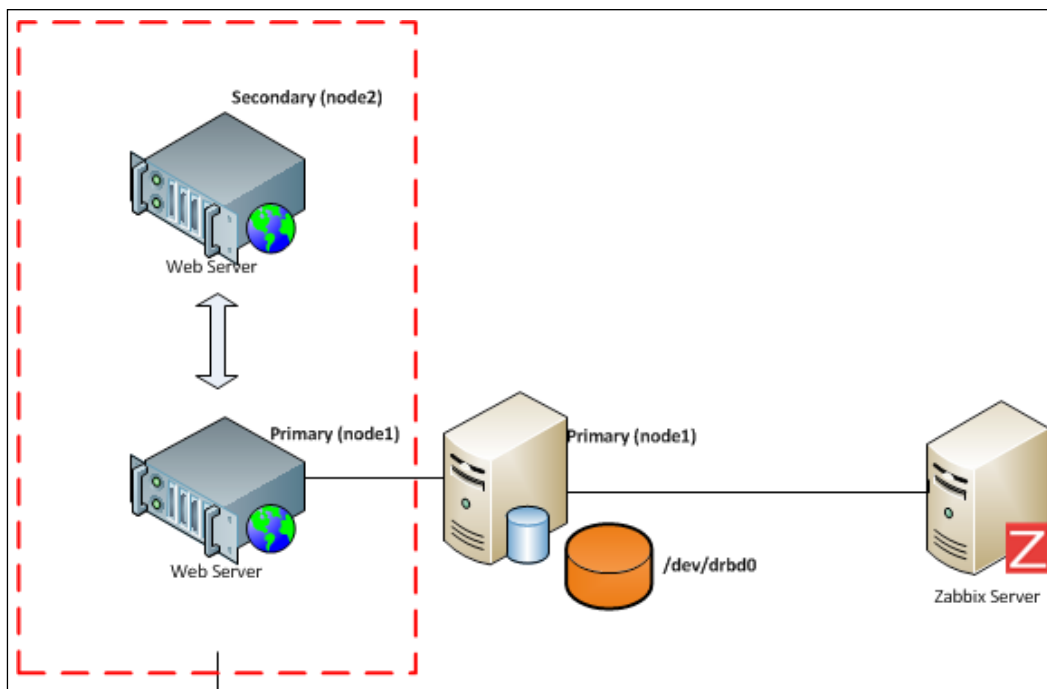
Now that you know all the software components in play, it's time to go deep into a web server HA configuration. This proposed design foresees two nodes with Apache on top that is bonded to a virtual IP address. In this design, the HTTPD or better Apache is on top of an active/passive cluster that is managed by Corosync/Pacemaker.

It is quite an easy task to provide a highly available configuration for the Zabbix GUI because the web application is well defined and does not produce or generate data or any kind of file on the web server. This allows you to have two nodes deployed on two different servers, if possible on two distant locations, implementing a highly available fault-tolerant disaster-recovery setup. On this configuration, since the web content will be "static", in the sense that it will not change (apart from in case of system upgrade), you don't need a filesystem replication between the two nodes. The only other component that is needed is a resource manager that will detect the failure of the primary node and will coordinate the failover on the secondary node. The resource manager that will be used is Pacemaker/Corosync.

The installation will follow this order:

1. Installation of HTTPD server on both nodes
2. Installation of Pacemaker
3. Deployment of the Zabbix web interface on both nodes
4. Configuration of Apache to bind it on VIP
5. Configuration of Corosync/Pacemaker
6. Configuration of the Zabbix GUI to access RDBMS (on VIP of PostgreSQL)

The next diagram explains the proposed infrastructure:



Configuring HTTPD HA

This software is a sophisticated cluster resource manager that is widely used with a lot of features. To set up Pacemaker, you need to:

- Install Corosync
- Install Pacemaker
- Configure and start Corosync

It is time to spend a couple of words on this part of the architecture. Corosync is a software layer that provides the messaging service between servers within the same cluster.

Corosync allows any number of servers to be a part of the cluster using different fault tolerant configurations such as Active-Active, Active-Passive, and N+1. Corosync, in the middle of its tasks, checks that Pacemaker is running and practically bootstraps all the process that is needed.

To install this package, you can run the following command:

```
$ yum install pacemaker corosync
```

Yum will resolve all dependencies for you; once everything is installed, you can configure corosync. The first thing to do is to copy the sample configuration file available at the following location:

```
$ cp /etc/corosync/corosync.conf.example /etc/corosync/corosync.conf
```

To configure Corosync, you need to choose an unused multicast address and a port:

```
$ export MULTICAST_PORT=4000
```

```
$ export MULTICAST_ADDRESS=226.94.1.1
```

```
$ export BIND_NET_ADDRESS=`ip addr | grep "inet " | grep brd | tail -n1 |  
awk '{print $4}' | sed s/255/0/`
```

```
$ sed -i.bak "s/.*mcastaddr:./mcastaddr:\ $MULTICAST_ADDRESS/g" /etc/  
corosync/corosync.conf
```

```
$ sed -i.bak "s/.*mcastport:./mcastport:\ $MULTICAST_PORT/g" /etc/  
corosync/corosync.conf
```

```
$ sed -i.bak "s/.*bindnetaddr:./bindnetaddr:\ $BIND_NET_ADDRESS/g" /etc/  
corosync/corosync.conf
```

Now, you need to tell Corosync to add the Pacemaker service and create the `/etc/corosync/service.d/pcmk` file with the following content:

```
service {  
  # Load the Pacemaker Cluster Resource Manager  
  name: pacemaker  
  ver: 1  
}
```

At this point, you need to propagate the files you just configured on node2:

```
/etc/corosync/corosync.conf  
/etc/corosync/service.d/pcmk
```

After that, you can start Corosync and Pacemaker on both nodes:

```
$ /etc/init.d/corosync start  
$ /etc/init.d/pacemaker start
```

Check the cluster status using the following command:

```
$ crm_mon
```

Examine the configuration using the following command:

```
$ crm configure show
```

Understanding Pacemaker and STONITH

Shoot The Other Node In The Head (STONITH) can introduce a weak point in this configuration; it can cause a split-brain scenario especially if servers are in two distant locations where there are a wide number of causes that can avoid the communication between them. The split-brain scenarios happen when each node believes that the other is broken and it is the first node. Then when the second reboot occurs, it shoots the first and so on. This is also known as STONITH death match.

There are basically three issues that can cause one node to STONITH the other:

- Nodes are alive but unable to communicate with each other
- A node is dead
- An HA resource failed to stop

The first cause can be avoided ensuring redundant communication paths and by handling the multicast properly. This involves the whole network infrastructure, and if you buy a network service from a vendor, you cannot expect safety or trust and multicasts will not be well managed. The second cause is obvious, and it is unlikely that the node causes the STONITH death match.

The third cause is not easy to understand. This can be clarified with an example. Basically, an HA resource is started on a node. If it is started, the resource will be monitored indefinitely; if the start fails, the resource will be started and stopped and then restarted in either the current node or the second node. If the resource needs to be stopped and the stop happens, the resource is restarted on the other node. Now if the stop fails, the node will be fenced STONITH because it is considered as the safe thing to do.



If the HA resource can't be stopped and the node is fenced, the worse action is killing the whole node. This can cause data corruption on your node especially if there is ongoing transactional activity and this needs to be avoided. It's less dangerous if the HA service is a resource such as an HTTP server that provides web pages (without transactional activity involved); however, this is not safe.

There are different ways to avoid the STONITH death match, but we want the proposed design to be as easy as possible to implement, maintain, and manage, and so the proposed architecture can live without the STONITH actor that can introduce issues if not managed well and configured.



Pacemaker is distributed with STONITH enabled. STONITH is not really necessary on a two-node cluster setup.

To disable STONITH, use the following command:

```
$ crm configure property stonith-enabled="false"
```

Pacemaker – is Quorum really needed?

Quorum refers to the concept of voting; it means each node can vote what can happen. This is similar to democracy where the majority wins and implements decisions. For example, if you have a three-node or more cluster and one of the nodes on the pool fails, the majority can decide to fence the failed node.

With the Quorum configuration, you can also decide the no-Quorum-policy; this policy can be used for following purposes:

- **Ignore:** No action is taken if a Quorum is lost
- **Stop** (default option): It stops all resources on the affected cluster node
- **Freeze:** It continues running all the existing resources but doesn't start the stopped ones
- **Suicide:** It can fence all nodes on the affected partition

All these considerations are valid if you have a three-node or more configuration. Quorum is enabled by default on most configurations, but this can't apply on the two-node clusters because there is no majority to elect the winner and get a decision.

The following command needs to be disabled to apply the ignore rule:

```
$ crm configure property no-quorum-policy="ignore"
```


Pacemaker – the stickiness concepts

It is obviously highly desirable to prevent healthy resources being moved around the cluster. Moving a resource always requires a period of downtime that can't be accepted for critical service (such as the RDBMS), especially if the resource is healthy. To address this, Pacemaker introduces a parameter that expresses how much a service prefers to stay running where it is actually located. This concept is called **stickiness**. Every downtime has its cost, not necessarily represented by an expense that is tied to the little downtime period needed to switch the resource to the other node.

Pacemaker doesn't calculate this cost associated with moving resources and will do so to achieve the optimal resource placement.



On a two-node cluster, it is important to specify the stickiness; this will simplify all the maintenance tasks. Pacemaker can't decide to switch the resource to a maintenance node without disrupting the service.

Note that the Pacemaker's optimal resource placement does not always agree with what you would want to choose. To avoid this movement of resources, you can specify a different stickiness for every resource.

```
$ crm configure property default-resource-stickiness="100"
```



Is it possible to use INFINITY instead of a number on the stickiness properties? This will force the cluster to stay on that node until it's dead, and once the INFINITY node comes up, all will migrate back to the primary node.

```
$ crm configure property default-resource-stickiness="INFINITY"
```

Pacemaker – the Apache/HTTPD configuration

The Pacemaker resource manager needs access to the Apache server's status to know the status of HTTPD. To enable the server's status, you need to change the `httpd.conf` file as follows:

```
<Location /server-status>
  SetHandler server-status
  Order deny,allow
  Deny from all
  Allow from
127.0.0.1 <YOUR-NETWORK-HERE>/24
</Location>
```



For security reasons, it makes sense to deny the access to this virtual location and permit only your network and the localhost (127.0.0.1).

This kind of configuration foresees two web servers that will be called `www01` and `www02` to simplify the proposed example. Again, to keep the example as simple as possible, you can consider the following addresses:

- `www01 (eth0 192.168.1.50 eth1 10.0.0.50)`
- `www02 (eth0 192.168.1.51 eth1 10.0.0.51)`

Now, the first step is to configure the virtual address using the following commands:

```
$ crm configure
crm(live)configure#
primitive vip ocf:heartbeat:IPaddr2 \
> params ip="10.0.0.100"
# not that 10.0.0.100 is the
> nic="eth1" \
> cidr_netmask="24" \
> op start interval="0s" timeout="50s" \
> op monitor interval="5s" timeout="20s" \
> op stop interval="0s" timeout="50s"
crm(live)configure#
show
# make sure
node www01.domain.example.com
node www02.domain.example.com
primitive vip ocf:heartbeat:IPaddr2 \
    params ip="10.0.0.100" nic="eth1" cidr_netmask="24" \
    op start interval="0s" timeout="50s" \
    op monitor interval="5s" timeout="20s" \
    op stop interval="0s" timeout="50s"
property $id="cib-bootstrap-options" \
    dc-version="1.1.2-f059ec7cedada865805490b67ebf4a0b963bccfe" \
    cluster-infrastructure="openais" \
    expected-quorum-votes="2" \
```

```
no-quorum-policy="ignore" \  
stonith-enabled="false"  
rsc_defaults $id="rsc-options" \  
resource-stickiness="INFINITY" \  
migration-threshold="1"
```

```
crm(live)configure# commit  
crm(live)configure# exit
```

Using `commit`, you can enable the configuration. Now, to be sure that everything went well, you can check the configuration using the following command:

```
$ crm_mon
```

You should get an output similar to the following one:

```
=====  
Last updated: Fri Jul 12 10:59:16 2013  
Stack: openais  
Current DC: www01.domain.example.com - partition WITHOUT quorum  
Version: 1.1.2-f059ec7cedada865805490b67ebf4a0b963bccfe  
2 Nodes configured, unknown expected votes  
1 Resources configured.  
=====
```

```
Online: [ www01.domain.example.com www02.domain.example.com ]
```

```
vip (ocf::heartbeat:IPaddr2): Started www01.domain.example.com  
To be sure that the VIP is up and running you can simply ping it  
$ ping 10.0.0.100
```

```
PING 10.0.0.100 (10.0.0.100) 56(84) bytes of data.  
64 bytes from 10.0.0.100: icmp_seq=1 ttl=64 time=0.012 ms  
64 bytes from 10.0.0.100: icmp_seq=2 ttl=64 time=0.011 ms  
64 bytes from 10.0.0.100: icmp_seq=3 ttl=64 time=0.008 ms  
64 bytes from 10.0.0.100: icmp_seq=4 ttl=64 time=0.021 ms
```

Now you have the VIP up and running. To configure Apache in the cluster, you need to go back on the CRM configuration and tell Corosync that you will have a new service, your HTTPD daemon, and will have to group it with the VIP. This group will be called webserver.

This configuration will tie the VIP and the HTTPD and both will be up and running on the same node. We will configure the VIP using the following commands:

```
$ crm configure
crm(live)configure# primitive httpd ocf:heartbeat:apache \
> params configfile="/etc/httpd/conf/httpd.conf" \
> port="80" \
> op start interval="0s" timeout="50s" \
> op monitor interval="5s" timeout="20s" \
> op stop interval="0s" timeout="50s"
crm(live)configure# group webserver vip httpd
crm(live)configure# commit
crm(live)configure# exit
```

Now, you can check your configuration using the following commands:

```
$ crm_mon
=====
Last updated: Fri Jul 12 11:03:50 2013
Stack: openais
Current DC: www01.domain.example.com - partition WITHOUT quorum
Version: 1.1.2-f059ec7cedada865805490b67ebf4a0b963bccfe
2 Nodes configured, unknown expected votes
1 Resources configured.
=====
```

```
Online: [ www01.domain.example.com www02.domain.example.com ]
```

```
Resource Group: webserver
```

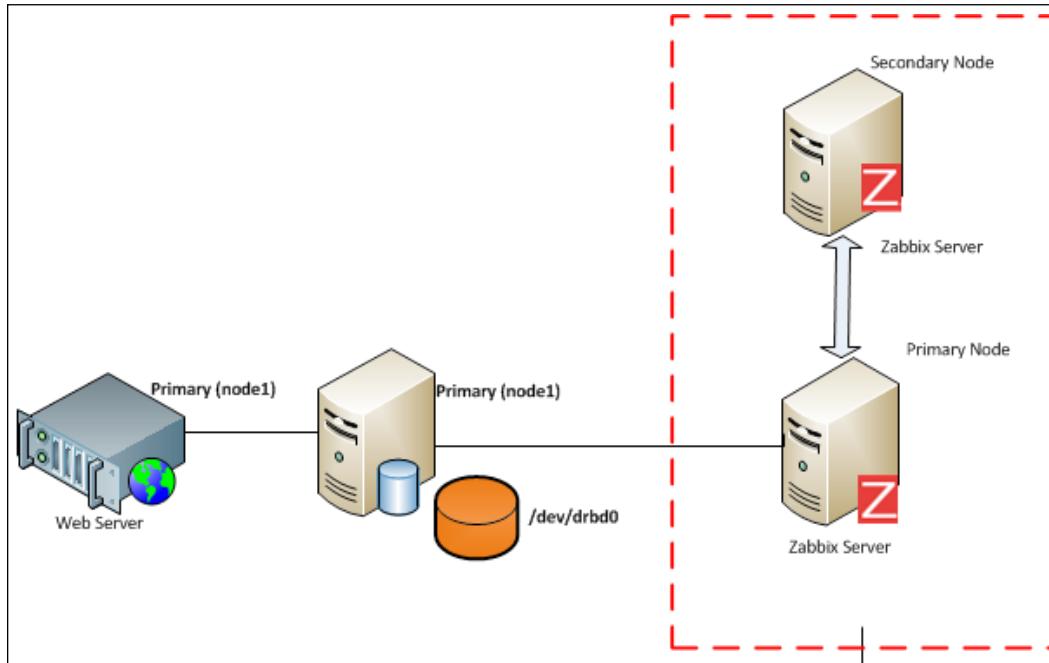
```
      vip      (ocf::heartbeat:IPaddr2):      Started www01.domain.
example.com
      httpd     (ocf::heartbeat:apache):      Started www01.domain.
example.com
```



Note that since you are not using Quorum, make sure that the `crm_mon` display: `partition WITHOUT quorum` and `unknown expected votes` are normal.


Configuring the Zabbix Server for high availability

A high availability cluster for a Zabbix server is easier to configure compared to Apache or a database server. Whether it's a standalone server or a node that is a part of a distributed setup, the procedures are exactly the same.



Once you have installed Corosync and Pacemaker on the two nodes (see the previous sections for details), you will also install Zabbix on the nodes that will make the cluster. You will then need to configure Zabbix to listen to the virtual IP that you have identified for the cluster. To do so, change both `SourceIP` and `ListenIP` to the appropriate value in the `zabbix_server.conf` configuration file:

```
SourceIP=10.10.1.9
ListenIP=10.10.1.9
```

 Needless to say, change the IP value to the one that is appropriate for your environment and you have reserved a virtual IP for the Zabbix cluster.

You can now proceed to disable STONITH using the following command:

```
$ crm configure property stonith-enabled="false"
```

If you have just two nodes, you also need to disable the quorum, otherwise the cluster wouldn't know how to obtain a majority:

```
$ crm configure property no-quorum-policy="ignore"
```

And finally, set the service stickiness high enough so that you don't have a service going back and forth between the nodes and it stays where it is, unless you manually move it or the active node goes down:

```
$ crm configure property default-resource-stickiness="100"
```

Much like the Apache/HTTPD cluster configuration, you now need to define a primitive for the virtual IP:

```
$ crm configure primitive Zbxvip ocf:heartbeat:IPaddr2 \  
params ip="10.10.1.9" iflabel="httpvip" \  
op monitor interval="5"
```

For the Zabbix server, define primitive using the following command:

```
$ crm configure primitive Zabbix lsb::zabbix_server \  
op monitor interval="5"
```

Just like the previous section, all that is now left to do is to group the primitives together, set up colocation, service StartOrder, and you are done:

```
$ crm configure group Zbx_server Zbxvip Zabbix meta target-role="Started"  
$ crm configure colocation Ip_Zabbix inf: Zbxvip Zabbix  
$ crm configure order StartOrder inf: Zbxvip Zabbix
```

As you can see, the simpler the components, the easier it is to set them up in a cluster configuration using Pacemaker. While it is still fairly easy and simple, things start to change when you turn to configure the most critical part of any high availability setup: database and data storage.

Database high availability

Implementing high availability for a database is not an easy task. There are a lot of ways to implement a high availability configuration using different software and complexity.

The architecture proposed here is fully redundant; it is one of the possible solutions that is widely used on large environments. You need two database servers to implement this solution, and two installations of the same software and operating system. Obviously, since servers are twins and are tied together, they need to have the same software, the same release patch, and basically be identical.

Since we are going to have two different servers, it is clear that the data needs to be replicated between them; this implies that your server needs to be interconnected with a dedicated network connection that is capable of providing the needed throughput.

In this design, your server can be placed on the same location or located at two different data centers that provide a reliable disaster-recovery solution. In this case, we are going to provide a highly available design.

There is a different way to provide data replication between two servers. They are as follows:

- File system replication
- Shared disk failover
- Hot/warm standby using PITR
- Trigger-based master-standby replication
- Statement-based replication middleware
- Asynchronous multimaster replication
- Synchronous master replication

There are positive and negative sides to each one of them. Between all these options, we can exclude all the solutions that are trigger-based because all of them introduce an overhead on the master node. Also, adding a user-lever layer can be imprecise/inexact.

Between these options, there are few solutions that permit having a low or really low mean time to recovery and are safe from data loss. The solutions that guarantee that in case of master failure there will no data loss are as follows:

- Shared disk failover
- Filesystem replication
- Statement-based replication

A solution that adopts a shared disk failover cluster implies the usage of a shared san. This means that if we want to place your server on a separate server farm, on a different location, this system will be really expensive.

If the solution adopts a warm and hot standby using a **point-in-time recovery (PITR)** and your node goes down, you need enough free space to handle and store all the transaction logfiles generated. This configuration by design needs a secondary database (identical to the master node) that is a warm standby and waits for the log transaction. Once arrived, the RDBMS needs to apply the transaction on your secondary node.

In this case, if secondary node goes down, we need to be warned because the primary database will produce the archive logfiles that are not shipped, and this can bring your infrastructure to a halt. On a large environment, the transactional activity is normally heavy, and if the fault happens to be out of the normal working hours, this HA configuration needs to be handled.

Another way is the PostgreSQL synchronous replication. If the secondary node goes down, this configuration would need a reload to prevent hanging of the transaction.

The trigger-based configurations are heavy and dangerous because they imply that a trigger can go on firing every insert, and replicate the same insert on the secondary node introducing a feasible overhead. Partitioning with inheritance is not well supported by this method. Also, this method does not gives us warranty against data loss when the master fails.

Infrastructures that include a second standby database introduce a second actor, that is, if it is down or unreachable, it shouldn't cause a master to hang. Nowadays, with PostgreSQL 9.1, a synchronous replication is a viable solution. These configurations unfortunately add some constraints: the transmission must be acknowledged before the commit happens, and the transmission doesn't guarantee that you will have a reply.

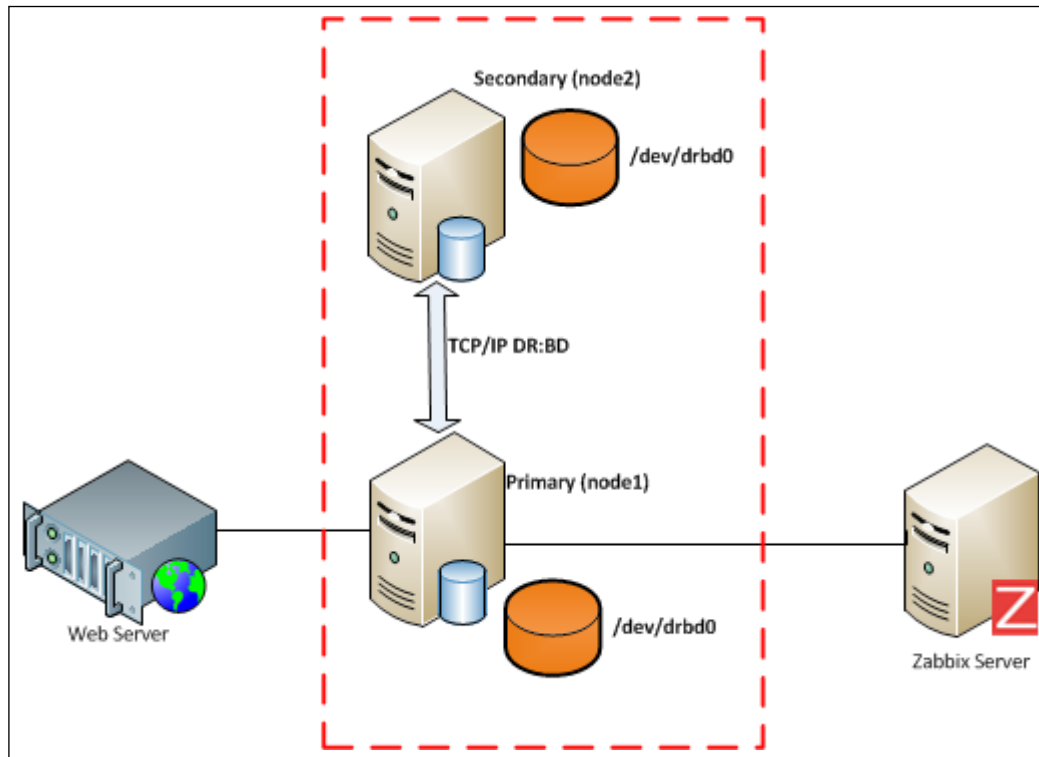
This practically means that if the secondary node goes down, the primary database will hang until the slave receives the transaction and notifies back to the master that this one has been acquired. The result is that a primary node can hang for an indefinite time and this practically doubles the risk of downtime.

An issue on slaves node shouldn't impact the primary node. This practically doubles the risk of downtime and is not acceptable in the context of high availability.

Clustering of PostgreSQL

The cluster presented here is simple and is designed to have as few actors in play as possible, but with the high availability design in mind.

The architecture shown in the following diagram is efficient. It has a minimum number of actors in play and is easy to monitor, maintain, and upgrade.



Mirrored logical volume with LVM and DRBD

LVM2 is the Linux implementation of the **logical volume manager** (LVM) on the Linux logical device mapper framework. This LVM2, apart from the name, doesn't have anything in common with the previous one.

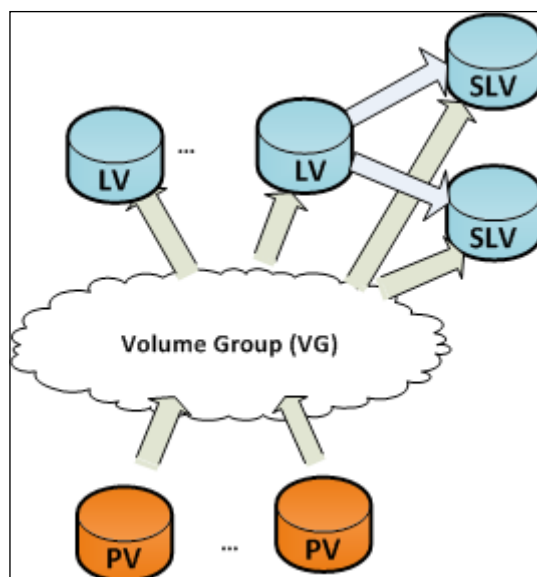
The basic concepts of LVM2 are as follows:

- **Physical Volume (PV):** This is the actual physical partition or storage system on which the LVM system is built.

- **Volume Group (VG):** A volume group is the basic administrative unit. It may include one or more PV. Every VG has a unique name and can be extended at runtime by adding more additional PVs or enlarging the existing PV.
- **Logical Volume (LV):** This is available as a regular block device to the Linux kernel and its components can be created at runtime within the available volume groups. Logical volumes can be resized when online and also moved from one PV to another PV if they are on the same Volume Group.
- **Snapshot Logical Volume (SLV):** This is a temporary point-in-time copy of LVs. The strong point is that if the size is really big (several hundred gigabytes), the space required is significantly less than the original volume.



The partition type Linux LVM that owns the signature **0x8E** is used exclusively for LVM partition. This however is not required. LVM indeed recognizes the physical volume group by a signature written on the PV initialization.



Since a logical volume once created is simply seen as a block device, you can use DRBD on it.

Prerequisite tasks to start with DRBD on LVM

While setting up DRBD over LVM, there are some basic steps to bear in mind:

- LVM needs to know about your DRBD devices
- LVM caching needs to be disabled
- Remember to update the initramfs with the new kernel device map

LVM by default scans all block device found on `/dev` looking for PV signatures; hence, we need to set an appropriate filter on `/etc/lvm/lvm.conf`:

```
filter = ["a|sd.*|", "a|drbd.*|", "r|.*/"]
```

This filter accepts all the SCSI disk plus DRBD disks. Now, we need to rescan all your volume groups with the following command:

```
# vgscan
```

It is important that you remember to disable the LVM caching because DRBD disks will disappear in case of a failure. This is normal when we face a fault, and if caching is not disabled, it is possible that you will see the disk is available when in reality it is not.

This is done by adding the following line on `/etc/lvm/lvm.conf`:

```
write_cache_state = 0
```

Now that the cache has been disabled, it is possible that we still have some portion or piece of cache on your disks that were previously generated. We need to clean up the following location:

```
/etc/lvm/cache/.cache
```

Now, it's better to regenerate the kernel device map files with the following command:

```
# update-initramfs -u
```

Now, it is possible for us to go ahead with the configuration.

Creating a DRBD device on top of the LVM partition

Now that your caching is disabled and the LVM is properly configured, we need to create your PV. To initialize your SCSI partitions as physical volume, we run the following commands from the root:

```
$ pvcreate /dev/sda1
   Physical volume "/dev/sda1" successfully created
$ pvcreate /dev/sdb1
   Physical volume "/dev/sdb1" successfully created
```

The given output tells us that the volume has been initialized. Now you can create a low-level VG, `vgpgdata`:

```
$ vgcreate vgpgdata /dev/sda1 /dev/sda2
   Volume group "vgpgdata" successfully created
```

Finally, you can create your volume or a better logical volume that will be used as the DRBD's block device:

```
$ lvcreate --name rpgdata0 --size 10G local
   Logical volume "rpgdata0" created
```

All these steps need to be repeated in the same order on both your nodes. Now you need to install DRBD on both nodes using the following command:

```
$ yum install drbd kmod-drbd
```



To install the DRBD, it is important to have the **EXTRAS** repositories enabled.

Now edit the `drbd.conf` file located at `/etc/drbd.conf` and create the `rpgdata0` resource as follows:

```
resource rpgdata0 {
    device /dev/drbd0;
    disk /dev/local/rpgdata0;
    meta-disk internal;
    on <host1> { address <address_host1>:<port>; }
    on <host2> { address <address_host2>:<port>; }
}
```



Replace `host1`, `host2`, `address_host1`, and `address_host2` with the two hostnames and their respective network addresses.

Be sure to have copied the `drbd.conf` file on both nodes before proceeding with the next section. Disable automatic start for DRBD because it will be managed by Pacemaker.

```
$ chkconfig drbd off
```

Enabling resource in DRBD

Now that the configuration file has been copied on all your nodes, you can finally initialize the device and create the required metadata.

This initialization process needs to be executed on both nodes and can be run from the root using the following command:

```
$ drbdadm create-md rpgdata0
v08 Magic number not found
Writing meta data...
initialising activity log
NOT initializing bitmap
New drbd meta data block successfully created.
```



This is the initialization process and needs to be executed only on a new device.

Now you can enable the `rpgdata0` resource:

```
$ drbdadm up rpgdata0
```

The process can be observed by looking at the `/proc` virtual filesystem:

```
$ tail /proc/drbd
version: 8.4.1 (api:1/proto:86-100)
GIT-hash: 91b4c048c1a0e06837625f65d312b38d47abara80 build by buildsysteem@
linbit, 2013-02-20 12:58:48

 0: cs:Connected ro:Secondary/Secondary ds:Inconsistent/Inconsistent C
r-----

    ns:0 nr:0 dw:0 dr:0 al:0 bm:0 lo:0 pe:0 ua:0 ap:0 ep:1 wo:b
oos:524236
```



The Inconsistent/Inconsistent state here at this point is normal. You need to specify which node is the master and which will be the source of this synchronization.

At this point, DRBD has allocated disk and network and is ready to begin the synchronization.

Defining a primary device in DRBD

The primary promotion is quite easy; you need go to the primary node and run this command:

```
$ drbdadm primary rpgdata0
```

Now the server on which you run this command becomes the master of the replication server, and you can create the PV on that new device. So, on the master node, you need to run the following command:

```
$ pvcreate /dev/drbd0
```

```
Physical volume "/dev/drbd0" successfully created
```

Create your VG, which in this example will be `secured_vg_pg`:

```
$ vgcreate secured_vg_pg /dev/drbd0
```

```
Volume group "secured_vg_pg" successfully created
```

Finally, it is possible to create an LV on that PV using the following command:

```
$ lvcreate -L 6G -n secured_lv_pg secured_vg_pg
```

In this example, we reserved a space for snapshots; so, if you ever want one, you have enough space for that. Finally, it is possible to set up the filesystem.

Creating a filesystem on a DRBD device

Now it is important to check whether the DRBD service is disabled from the startup and shutdown list because this service will be managed directly from Pacemaker. Once you disable the service, it is possible to create the filesystem on the new device, but before that it is important to do the following:

- Create a mountpoint
- Create a filesystem
- Mount the filesystem and make it available

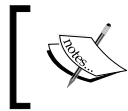
You can create your own mountpoint, but this step-by-step installation will use `/db/pgdata`:

```
$ mkdir -p -m 0700 /db/pgdata
```

Now, there are different filesystems supported by most of the distributions; RedHat 6.0 completely supports XFS. XFS has an important feature that permits parallel access to filesystem. It supports parallel read/write. XFS allows us to write the same files from multiple threads concurrently; this obviously is a big improvement for large database tables and it also reduces the contention on filesystems.

To install XFS and the relative utils, use the following command:

```
$ yum install xfsprogs
```



XFS allows write access to the same file for multiple threads concurrently; this is interesting especially on DRBD usage where the contention on filesystems becomes an important factor.

Once installed and available, you can format the logical volume using the following command:

```
$ mkfs.xfs /dev/secured_vg_pg/secured_lv_pg
```



Once created, the filesystem can't be reduced but only enlarged using the `xfs_growfs` command.

Now you can mount the filesystem using the following command:

```
$ mount -t xfs -o noatime,nodiratime,attr2 /dev/secured_vg_pg/secured_lv_pg /db/pgdata
```



Do not forget to add this partition on automount (`fstab`), otherwise you will lose your partition after a reboot.

All can be changed to your PostgreSQL process owner, usually `postgres`:

```
$ chown postgres:postgres /db/pgdata
```

```
$ chmod 0700 /db/pgdata
```



The filesystem creation steps need to be done only on the primary node.

Now the filesystem is mounted, formatted, and ready for PostgreSQL.

Pacemaker cluster – integrating DRBD

Pacemaker makes DRBD extremely powerful in a really wide variety of scenarios. There are some notable points that have already been discussed when we presented Pacemaker/Corosync. These points are as follows:

- Disable STONITH
- Disable Quorum
- Enable stickiness

As discussed earlier in this chapter, it is really important to avoid split-brain scenarios and STONITH death match. Just as a reminder, to disable STONITH, you can run the following command:

```
$ crm configure property stonith-enabled="false"
```

Since this again is a two-node cluster, it is strongly advised to disable Quorum. The command that permits us to do this is as follows:

```
$ crm configure property no-quorum-policy="ignore"
```

Now, it is preferred to enable stickiness. This argument has been discussed earlier in the chapter. Anyway, as a quick reminder, we can say that by enabling the stickiness, we have a guarantee of a preferred node over another. This will help you to keep your cluster on one side and have a preferred site where everything should run. The command for this is as follows:

```
$ crm configure property default-resource-stickiness="100"
```

Enabling the DRBD configuration

This section explains how to enable the DRBD-backend service in your Pacemaker cluster. There are some steps to be followed:

- Add DRBD to Pacemaker
- Add and define the master/slave resource

You need to have a master/slave resource that controls which node is primary and which one is secondary. This can be done with the following command:

```
$ crm configure primitive drbd_pg ocf:linbit:drbd \  
params drbd_resource="rpgdata0" \  
op monitor interval="15" \  
op start interval="0" timeout="240" \  
op stop interval="0" timeout="120"
```


Once done, you need to set up a resource that can promote or demote the DRBD service on each node. Keep in mind that the service needs to run on both the nodes at all times with a different state, defining a master/slave resource as follows:

```
$ crm configure ms ms_drbd_pg drbd_pg \  
meta master-max="1" master-node-max="1" clone-max="2" \  
clone-node-max="1" notify="true"
```

Pacemaker – the LVM configuration

Now you need to configure Pacemaker to:

- Manage the LVM
- Manage the filesystem

Because of the design and working of DRBD, the actual active volume will be invisible on the secondary node. On the secondary node, you can't mount or handle this volume. Having said that, you need to help DRBD to find devices:

```
$ crm configure primitive pg_lvm ocf:heartbeat:LVM \  
params volgrpname="secured_vg_pg" \  
op start interval="0" timeout="30" \  
op stop interval="0" timeout="30"
```

With the preceding configuration, Pacemaker will search for usable volume on DRBD devices and will be available using the DRBD resource promotion. Since the adopted filesystem on DRBD device is XFS, you need to define how to mount and handle this device:

```
$ crm configure primitive pg_fs ocf:heartbeat:Filesystem \  
params device="/dev/secured_vg_pg/secured_lv_pg" directory="/db/pgdata" \  
options="noatime,nodiratime" fstype="xfs" \  
op start interval="0" timeout="60" \  
op stop interval="0" timeout="120"
```

Since LVM is the last layer on this configuration, you can take advantage of snapshot capabilities and a good level of isolation

Pacemaker – configuring PostgreSQL

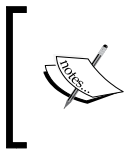
Now you can add the PostgreSQL configuration to the cluster.



PostgreSQL installation is not covered here because it is already discussed in *Chapter 1, Deploying Zabbix*.

The following lines add a primitive to the Pacemaker that will set a PostgreSQL health check every 30 seconds and define a timeout of 60 seconds to retrieve the response:

```
$ crm configure primitive pg_lsb lsbd:postgresql \
op monitor interval="30" timeout="60" \
op start interval="0" timeout="60" \
op stop interval="0" timeout="60"
```



This command extends the start and stop timeout because it will handle quite big databases. It can also happen that a Pacemaker may need to give the time to complete a checkpoint on shutdown and a recovery on startup.

Pacemaker uses those parameters in a primary manner to determine if PostgreSQL is available or not.

Pacemaker – the network configuration

Up until now, you haven't yet configured a predefined IP address for PostgreSQL. Since it doesn't make sense to have different addresses in case of a switchover or failover, you need to set up a virtual IP that will follow your service. This prevents any change of configuration for all your clients. You can use a cluster name or an IP address. For that, you need to issue the following line:

```
$ crm configure primitive pg_vip ocf:heartbeat:IPaddr2 \
params ip="192.168.124.3" iflabel="pgvip" \
op monitor interval="5"
NOTE: change the address 192.168.124.3 with your own.
```

Here it is not specified that the ARP address, `IPaddr2`, will automatically send five ARP packets, and this value can be increased if necessary.

Pacemaker – the final configuration

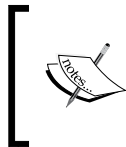
Now you have all the required components ready to be tied together in a group that will contain all your resources. The group is PGServer:

```
$ crm configure group PGServer pg_lvm pg_fs pg_lsb pg_vip
$ crm configure colocation col_pg_drbd inf: PGServer ms_drbd_pg:Master
```

The Master server specifies that your PGServer group depends on the master/slave setup reporting a master status that happens exclusively on an active mode. It is also true that the PGServer group depends on the DRBD master.

Now, it is important to specify the right order to start and shutdown all the services. We will use the following command to do so:

```
$ crm configure order ord_pg inf: ms_drbd_pg:promote PGServer:start
```



The `:promote` and `:start` are fundamentals, they mean that once `ms_drbd_pg` is promoted, the PGServer will start. With this precise order of events, if you omit the `:start`, Pacemaker can choose the start/stop order, and it might end in a broken state.

Cluster configuration – the final test

Finally, the cluster is ready! What to do next? It is simple! You can break your own cluster, play with the configuration, and check that all is fine before we go live with this new infrastructure.

Faults that need to be tested are as follows:

- The node goes offline
- Manual failover of the cluster
- Primary crash
- Secondary crash
- Forceful synchronization of all the data

Run the following command:

```
$ crm node standby HA-node2
```

If all is fine, `crm_mon` will respond with the following:

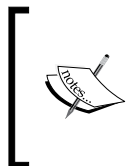
```
Node HA-node2: standby
Online: [ HA-node1 ]
```

You can easily fix this state by firing the following command:

```
$ crm node online HA-node2
```

Until here it is quite easy. Now you can try a failover of the whole cluster using the following command:

```
$ crm resource migrate PGServer HA-node2
```



You can migrate PGServer to the second node. If that becomes unavailable, Pacemaker will move to the primary node until the secondary return. This is because the `migrate` command will give a higher score to the named node, and this will win on your specified stickiness.

The server can be migrated back with the following:

```
$ crm resource unmigrate PGServer
```

Now, you can switch off the secondary node and Pacemaker will respond with the following:

```
Online: [ HA-node1 ]
OFFLINE: [ HA-node2 ]
Master/Slave Set: ms_drbd_pg [drbd_pg]
Masters: [ HA-node1 ]
Stopped: [ drbd_pg:1 ]
```

After that, you can start up the secondary node again. Now, switch off the secondary node and Pacemaker will respond with the following:

```
Online: [ HA-node1 HA-node2 ]
Master/Slave Set: ms_drbd_pg [drbd_pg]
Masters: [ HA-node1 ]
Slaves: [ HA-node2 ]
```

Now as a final test, you can invalidate all the data on the secondary node with the following command:

```
$ drbdadm invalidate-remote all
```

Or from the secondary node, you can run the following command:

```
$ drbdadm invalidate all
```

This will force DRBD to consider all the data on the secondary node out of sync. Therefore, DRBD will resync all the data on the secondary node before getting it from the primary node.

DRBD performance and optimizations

There are some aspects that can be improved and that should be considered when you implement a DRBD cluster. There are some optimizations that can be applied. You need to consider that if your database, or more in general, the second node of the DRBD cluster is on a different location that is far away from your datacenter, the network bandwidth can have an efficient synchronization, which plays a fundamental role. Another thing that needs to be considered on a disaster recovery site is the bandwidth and its cost. It is also important to calculate and understand how much data is required and the transfer rate that we can reach or need.

Efficient synchronization by DRBD

Synchronization is a distinct process and can't be considered like device replication. While replication happens only the first time you start up the device, synchronization, and resynchronization as well, is decoupled from incoming writes. On the proposed architecture, synchronization is necessary when:

- The link has been interrupted
- The Server had a fault on the primary node
- The Server had a fault on the secondary node

DRBD doesn't synchronize blocks sequentially and not in the order they were originally written.



While synchronization is on-going, during the process, on the disk you will have partly obsolete data and partly updated data.

The service will continue to run on the primary node while background synchronization is in progress. Since this configuration has a LVM layer on top of DRBD, it is possible to use snapshots during the synchronization; this is a strong point of this architecture. While synchronization is ongoing, you are in a delicate phase because there is a single point of failure; only the primary is working fine, and if something happens here, you might completely lose all the data and the secondary node may contain bad data. This critical situation can be mitigated with the LVM snapshot.




Use of snapshot before beginning the synchronization can give you hands-on experience of that situation because data on secondary node are consistent, valid but not recently updated. Enabling snapshot before beginning synchronization will reduce the **Estimate Time to Repair (ETR)** also known as the **Recovery Time Objective (RTO)**.

To automate the snapshot, you can add the following lines on your DRBD configuration:

```
resource RESOURCE_NAME {
    handlers {
        before-resync-target "/usr/lib/drbd/snapshot-resync-target-lvm.
sh";
        after-resync-target "/usr/lib/drbd/unsnapshot-resync-target-lvm.
sh";
    }
}
```

The `snapshot-resync-target-lvm.sh` script is called before we begin the synchronization and the `unsnapshot-resync-target-lvm.sh` script will remove the snapshot once synchronization completes.


 If the script fails, the synchronization will not commence.

To optimize the synchronization DRBD support, a checksum-based synchronization is required. A checksum-based synchronization is more efficient in the sense that a brute force overwrites and blocks synchronization; this is not enabled by default. With these features enabled, DRBD reads blocks before synchronizing them and calculating a hash of the contents. It compares the hash calculated with the same data obtained from the same sector on the out-of-sync secondary, and if hash matches, DRBD omits to rewrite these blocks.

To enable this feature, you need to add the following lines on the DRBD configuration:

```
resource <resource>
    net {
        csums-alg <algorithm>;
    }
    ...
}
```

The `<algorithm>` tag is any message digest supported by the kernel crypto API, usually one among `sha1`, `md5`, and `crc32c`.

 If this change is done on an existing resource, you need to copy the changed `drbd.conf` file on the secondary client and afterwards run:

```
drbdadm adjust <resource>
```

Enabling online verification through DRBD

Online verification enables a block-by-block data integrity check in a very efficient way. This is particularly interesting for the efficiency on bandwidth usage plus it doesn't interrupt or break redundancy in any way.



Online verification is a CPU-intensive process; it will impact the CPU load.

DRBD with this functionality will calculate a cryptographic digest of every block on the first node, and then this hash is sent to the peer node that will do the same check. If the digest differs, the block will be marked out-of-sync and DRBD will retransmit only the marked blocks. This feature is not enabled by default and can be enabled by adding the following lines in `drbd.conf`:

```
resource <resource>
  net {
    verify-alg <algorithm>;
  }
  ...
}
```

Also, here `<algorithm>` can be any digest supported by crypto API usually by `sha1`, `md5`, and `crc32c`. Once configured, it is possible to run the online verification with the following command:

```
$ drbdadm verify <resource>
```



Since the check introduced will ensure that both nodes are perfectly in sync, it is advised to schedule a weekly or a monthly check within `crontab`.

If you have an out-of-sync block, it is possible to resync them simply with the following command:

```
$ drbdadm disconnect <resource>
$ drbdadm connect <resource>
```

DRBD – some networking consideration

When you use a block-based filesystem over a DRBD, it is possible to improve the transfer rate, enlarging the **Maximum Transmission Unit (MTU)** to higher values.

Block-based filesystem will have a noticeable improvement. Block-based filesystems are ext3, ReiserFS, and GFS. The filesystem proposed here on this architecture is extent-based and is not expected to see high improvement enabling the jumbo frame.

DRBD permits us to set up the synchronization rate. Normally, DRBD will try to synchronize the data on the secondary node as quickly as possible to reduce the inconsistent data time. Anyway, you need to prevent degrading of a performance that is caused by the bandwidth consumed for the synchronization.



Make sure you set up this parameter in relation to the bandwidth available; for instance, it doesn't make any sense to set up a rate that is higher than the maximum throughput.

The maximum bandwidth used from the background process of resynchronization is limited by a parameter: rate expressed in bytes; so, 8192 means 8 MiB. To fix the rate, you can change the DRBD configuration file by adding in the following code:


```
resource <resource>
  disk {
    resync-rate 50M;
    ...
  }
  ...
}
```



The rule to calculate the right rate and the resync rate is $\text{MAX_ALLOWED_BANDWIDTH} * 0.3$. It means we are going to use 30 percent of the maximum bandwidth available.

The sync rate follows exactly the same rule and can be specified as well on the `drbd.conf` file:

```
resource <resource>
  syncer {
    rate 50M;
    ...
  }
  ...
}
```


[ The syncer rate can be temporarily modified with the following command:

```
drbdsetup /dev/drbdnum syncer -r 120M
```

The resync rate can be temporarily changed with the following command:

```
drbdadm disk-options --resync-rate=110M <resource>
```

Both these rates can be reverted with the following command:

```
drbdadm adjust resource
```

]


DRBD give us other interesting parameters to finetune the system and optimize performances; of course, those that follow are not solutions to all the throughput issues. They can vary from system to system but it is useful to know that they exist and you can get some benefit from them.

In particular, there are two parameters. They are as follows:

- `max-buffers`
- `max-epoch-time`

The first properties (`max-buffers`) represent the maximum number of buffer DRBD. The second properties (`max-epoch-time`) represent the maximum number of write requests permitted between two write barriers. Both can be changed inside the `drbd.conf` file:

```
resource <resource> {  
    net {  
        max-buffers 8000;  
        max-epoch-size 8000;  
        ...  
    }  
    ...  
}
```

[ The default value for both is 2048, but they both can be changed to 8000. This is a reasonable value for most of the modern raid-SCSI controllers.]

There is another network optimization that can be done. Change the send buffer of the TCP/IP. By default, this value is set to 128 K, but if you are in a high-throughput network, such as a gigabit network, it make sense to increase this value to 512 K.

```
resource <resource> {  
    net {  
        sndbuf-size 512K;  
        ...  
    }  
    ...  
}
```



If you set these properties to 0, the DRBD will use the autotuning feature, adapting the TCP to send buffer to the network.

To close this optimization section, it is important to say that DRBD manages some other parameters:

- no-disk-barrier
- no-disk-flushes
- no-disk-drain

My personal advice is stay away from them if you don't really know what kind of hardware you have. Set them to represent a "big iron" on the system raid. These parameters disable the write barriers, the disk flush, and drain. Usually, all these features are managed directly from the controller. It doesn't make any sense to enable DRBD to manage them.

Summary

In this chapter, you have learned some fundamental concepts about high availability and service clustering. You have also learned how to apply them to the Zabbix server architecture using the open source Pacemaker service manager suite and the filesystem replication with DRBD. You should also have learned to value keeping things light and simple by choosing as few nodes as possible while maintaining a robust, redundant architecture. This completes the first part of the book that was focused on choosing the optimal Zabbix solution for an environment of any size. By choosing the right hardware and supporting software (refer to *Chapter 1, Deploying Zabbix* and *Chapter 2, Distributed Monitoring*) and high availability for the most sensitive components, you should now have a Zabbix installed that is perfectly tailored to your needs and environment.

In the rest of the book, we will focus on using this setup to actually monitor your network and servers and make use of the collected data beyond simple alerts. The next chapter will focus on data collection and using many of Zabbix's built-in item types to obtain monitoring data from a number of simple or complex or aggregated sources.

4

Collecting Data

Now that you have a Zabbix installation that is properly sized for your environment, you will want to actually start monitoring it. While it's quite easy to identify which hosts and appliances, physical or otherwise, you may want to monitor, it may not be immediately clear what actual measurements you should take on them. The metrics you can define on a host are called items, and this chapter will discuss their key features and characteristics. The first part will be more theoretical, and will focus on the following:

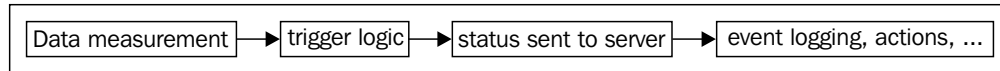
- Items as metrics, not status checks
- Data flow and directionality for items
- Trapper items as a means to control the data flow

We will then move to a more practical and specific approach and will discuss how to configure items to gather data from the following data sources:

- Databases and ODBC sources
- Java applications and the JMX console
- SNMP agents
- Web pages

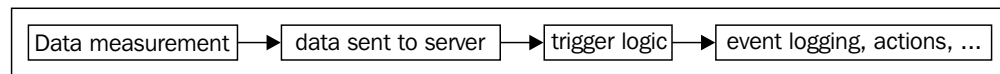
Gathering items as raw data

One of the most important features that sets Zabbix apart from most other monitoring solutions is that its main mode of interaction with the monitored objects is focused on gathering raw data, as opposed to alerts or status updates. In other words, many monitoring applications have the workflow (or variation) as shown in the following diagram:



That is, an agent or any other monitoring probe is asked to not only take a measurement, but also incorporate some kind of status decision about the said measurement before sending it to the main server's component for further processing.

On the other hand, the basic Zabbix workflow is subtly but crucially different, as shown in the following diagram:



Here, an agent or monitoring probe is tasked with just the measurement part, and then it sends the said measurement to the server component for storage, and eventually for further processing.

The data is not associated to a specific trigger decision (Pass/Fail, OK/Warning/Error, or any other variation), but is kept on the server as a single data point or measurement. Where applicable, that is, for numeric types, it's also kept in an aggregate and trending format as min, max, and average, over different periods of time. Keeping data separated from the decision logic, but all in a single place, gives Zabbix two distinct advantages.

The first one is that you can use Zabbix to gather data on things that are not directly related to the possible alerts and actions that you have to take, but related to the overall performance and behavior of a system. The classic example is that of a switch with many ports. You may not be interested in being alerted about anomalous traffic on every single port (as it may also be difficult to exactly define anomalous traffic on a single port with no contextual information), but you may be interested in gathering both port-level and switch-level traffic measurement in order to establish a baseline, evaluate possible bottlenecks, or plan for an expansion of your network infrastructure. Similar cases can be made about the CPU and core usage, storage capacity, number of concurrent users on a given application, and many more. At its simplest, Zabbix could even be used to gather the usage data, and visualize it in different graphs and plots, without even touching its powerful trigger and correlation features, and still prove to be an excellent investment of your time and resources.

Speaking of triggering, the second big advantage of having a full, central database of raw data as opposed to a single measurement (or at best, just a handful of measurements of the same item), is that for every trigger and decision logic need that you may have, you can leverage the whole measurements database to exactly define the kind of event that you want to monitor and be alerted on. You don't need to rely on a single measurement; you don't even need to rely on the latest measurement plus a few of the previous ones of the same item, or limit yourself to items from the same host. In fact, you can correlate anything with anything else in your item history database. This is a feature so powerful that we have dedicated an entire chapter to it, and you can go directly to *Chapter 6, Managing Alerts*, if that's what you want to read about. It would suffice to say that all this power is based on the fact that Zabbix completely separates its data-gathering functions from its trigger logic and action functions. All of this is based on the fact that measurements are just measurements, and nothing else.

So, in Zabbix, an item represents a single metric, a single source of data, and measurements. There are many kinds of native Zabbix items, even without considering the custom ones that you can define using external scripts. In this chapter, you will learn about some of the less obvious but very interesting ones. You will see how to deal with databases, how to integrate something as alien as SNMP traps to the Zabbix mindset, how to aggregate the existing items together to represent and monitor clusters, and more. As you lay a solid foundation, with sensible and strategic item definition and data gathering, you will be able to confidently rely on it to develop your event management and data visualization functions, as you will see in the following chapters.

Understanding the data flow for Zabbix items

A Zabbix item can be understood by its bare essentials – an identifier, data type, and associated host. These are the elements that are generally more useful for the rest of Zabbix's components. The identifier (that's usually the name and the associated item key) and the associated host are used to distinguish a single item among the thousands that can be defined in a monitoring environment. The data type is important so that Zabbix knows how to store the data, how to visualize it (text data won't be graphed, for example), and most importantly, what kind of functions can be applied to it in order to model triggers and the further processing.

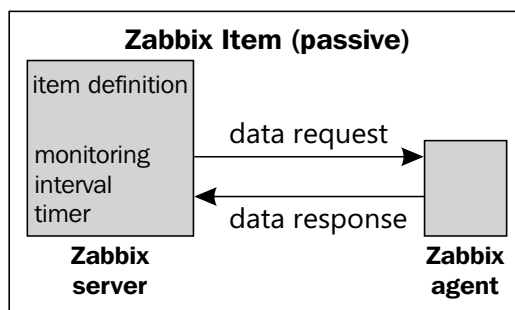


The item's name is a descriptive label that is meant to be easily read, while the item's key follows a specific syntax and defines exactly the metric that we want to measure.

Two other very important elements that are common for all the items are the history (and trends) retention period and item type. We already saw in *Chapter 1, Deploying Zabbix*, how history retention directly affects the size of the monitoring database, how to estimate it, and how to strike a balance between the performance and data availability. On the other hand, the item type is essential as it tells Zabbix how the item data is actually going to be made available to the server, in other words, how is Zabbix going to collect the data: through an agent, an SNMP query, an external script, and so on.

As you probably already know, there's a fair number of different item types. While it's fairly easy to understand the difference between an SSH item and an ODBC one, it's also important to understand how the data is passed around between the server and its probes, and whether they are a Zabbix agent, a server-side probe, or an external check of some kind. To this end, we'll first concentrate on the Zabbix agent and the difference between a passive and active item.

First of all, the active and passive concepts have to be understood from the agent's point of view, and not the server's. Furthermore, they serve to illustrate the component that initiates a connection in order to send or receive configuration information and monitor data.

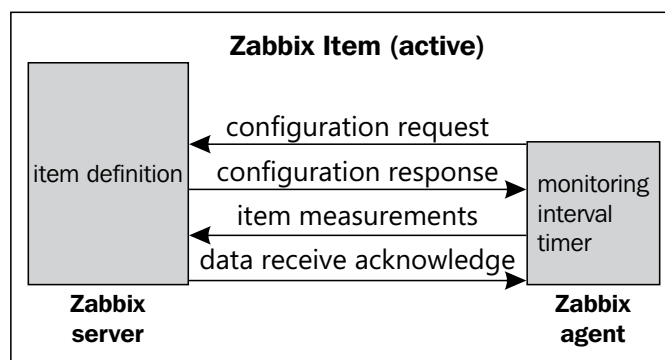


So, a standard Zabbix item is considered passive from the agent's point of view. This means that it's the server's job to ask the agent, at the time intervals defined for the item, to get the desired measurement and report it back immediately. In terms of network operations, a single connection is initiated and brought down by the server, while the agent is in the listening mode.

On the other end, in the case of a Zabbix active item, it's the agent's job to ask the server what monitoring data it should gather and at what intervals. It then proceeds to schedule its own measurements, and connects back to the server to send them over for further processing. In terms of network operations, the following are the two separate sessions involved in the process:

- The agent asks the server about items and monitoring intervals
- The agent sends the monitoring data it collected to the server

Unlike standard passive items, you'll need to configure an agent so that it knows which server it should connect to for the purpose of configuration and data exchange. This is of course defined in the `zabbix_agentd.conf` file for every agent; just set `ServerActive` as the hostname or IP address of your Zabbix server, and `RefreshActiveChecks` to the number of seconds the agent should wait before checking if there are any new or updated active item definitions.



Apart from the network connection initiation, the main difference between a passive and active item is that in the latter, it's impossible to define flexible monitoring intervals. With a passive item, you can define different monitoring intervals based on the time of the day and the day of the week. For example, you could check the availability of an identity management server every minute during office hours and every ten minutes during the night. On the other hand, if you use an active item, you are stuck with just one option for monitoring the intervals.

You may also have noticed a more-than-passing resemblance between the Zabbix active and passive items, and the functionality and features of a Zabbix active and passive proxy.

In fact, you can choose between the active and passive items in much the same way and for the same reasons as you chose between an active or passive proxy in *Chapter 2, Distributed Monitoring*, to offload some of the server's scheduling job and to work around restrictions and limitations of your network and routing or firewall configuration.

There is, of course, one main difference between proxies and agents. It's not the fact that a proxy can gather monitoring data from many different hosts, while an agent is theoretically limited to monitoring just the host it's installed on.

The main difference when it comes to data flow is that the mode of operation of a proxy is applied to all the hosts and items that the proxy manages. In fact, it doesn't care about the nature of the items a proxy has to monitor. However, when an active proxy gathers its data (whether with active or passive agent items, external scripts, SNMP, SSH, and so on), it will always initiate all connections to the server. The same goes for a passive proxy; it doesn't matter if all the items it has to monitor are all active agent items; it will always wait for the main server for updates on configuration and measurement requests.

On the other hand, an active or passive item is just one of the many other items. A host can be defined by a mix of active and passive items; so you can't assume that an agent will always initiate all its connections to the server; to do that, all of the items that rely on the agent have to be defined as active, including the future ones.

Understanding Zabbix trapper items

An extreme version of an active item that still relies on the Zabbix agent protocol is the Zabbix **trapper item**. Unique among all other item types, a trapper item does not have a monitoring interval defined at the server level. In other words, a server will know if a Zabbix trapper item is defined, its data type, the host it's associated with, and the retention period for both history and trends. But it will never schedule a check for the item nor pass the scheduling and monitoring interval information to any proxy or agent. So it's up to the specific probe to be scheduled in some way, and then send the information about the gathered data to the server.

Trapper items are, in some respect, the opposite of Zabbix's external checks from a data flow's point of view. As you probably already know, you define an external check item type when you want the server to execute an external script to gather measurements, instead of asking an agent (Zabbix, SNMP, or others). This can exact an unexpected toll on the server's performance as it has to fork a new process for every external script it has to execute, and then wait for the response. As the number of external scripts grows, it can significantly slow down the server operations to the point of accumulating a great number of overdue checks while it's busy executing scripts.

An extremely simple and primitive, yet effective way to work around this problem (after reducing the number of external scripts as much as possible, of course) is to convert all external check items to trapper items, schedule the execution of the same scripts used in the external checks through the crontab or any other scheduling facility, and modify the scripts themselves so that they use `zabbix_sender` to communicate the measured data to the server. When we talk about the Zabbix protocol in *Chapter 8, Handling External Scripts*, you'll see quite a few examples of this setup.

The data flow overview

This is a rundown of item types, classified with the connection type, with a proposed alternative in case you want, for any reason, to turn it around. As you can see, a Zabbix Trapper is often the only possible, albeit clunky or clumsy alternative, if you absolutely need to reverse a connection type. Note that in the following table, the term `Passive` means that the connection is initiated by the server, and `Active` means that the connection is initiated by whatever probe is used. While this may seem counter-intuitive, it's in fact coherent with the same terms as applied to proxies and agents.

Item Type	Direction	Alternative
Zabbix agent	Passive	Zabbix agent (active)
Zabbix agent (active)	Active	Zabbix agent
Simple check	Passive	Zabbix trapper
SNMP agent	Passive	Zabbix trapper
SNMP trap	Active	N/A
Zabbix internal	N/A (data about the server monitoring itself)	N/A (SNMP traps are completely different in nature)
Zabbix trapper	Active	Depends on the nature of the monitored data
Zabbix aggregate	N/A (uses data already available in the database)	N/A
External check	Passive	Zabbix trapper
Database monitor	Passive	Zabbix trapper
IPMI agent	Passive	Zabbix trapper
SSH agent	Passive	Zabbix trapper
TELNET agent	Passive	Zabbix trapper
JMX agent	Passive	Zabbix trapper
Calculated	N/A (uses data already in the database)	N/A

In the next few paragraphs, we'll delve more deeply into some of the more complex or interesting item types.

Database monitoring with Zabbix

Zabbix offers a way to query any database using SQL queries. The result retrieved from the database is saved as the item value, and can have, as usual, triggers associated with it. This functionality is useful in many applications. This gives you a way to monitor the user currently connected to a database, or the number of users connected to your web portal, or simply retrieve metrics from the DBMS engine.

Delving into ODBC

ODBC is a layer, a translation layer between **database management systems (DBMS)** and the application. Originally, it had been developed by Microsoft as a standard middleware for C programming language's API to handle the access to the DBMS. The applications use the ODBC function through the linked ODBC driver manager. This is basically the driver that passes the queries to the DBMS and dialogues to the ODBC. The ODBC layer is based on the ODBC device driver model, which basically encapsulates the logic needed to convert the standard command and function into specific calls. The ODBC driver has been implemented and developed from most of the DBMS vendors, to enable their database to interoperate with this layer. Most of the drivers don't implement all the functions defined in the standard, because not all of them are supported by all DBMS. The ODBC driver provides the application with a data source to interoperate with the database. The configuration file specifies the driver to load all the connections' parameters for each **Data Source Name (DSN)**, and all the DSNs are enumerated and defined inside this file. DSN also gives the functionality to present the entire database in a human-readable format. The DSN file needs to be protected. In the proposed setup, it is advisable to use a different Unix account for your Zabbix server, which will make things easy. As there is only one Zabbix server, the only user that needs to access this file is the Zabbix server user. This file should be owned by this user and be made unreadable to others. DSNs are contained in the `odbc.ini` file in the `etc` folder. This file will contain all the DSNs for all the different databases to which we want to connect. Take care to protect this file, and prevent other people from accessing this file because it can contain passwords.

There are two open source versions of ODBC available—**UnixODBC** and **iODBC**. Zabbix can use both of them, but before you can use it, the first thing to do is enable Zabbix to use ODBC and install the UnixODBC layer. There are two ways to do that: one is with the package manager, and the other one is to go through the old way of downloading and compiling it from the source (currently, the latest stable version is 2.3.1):

```
$ wget ftp://ftp.unixodbc.org/pub/unixODBC/unixODBC-2.3.1.tar.gz
$ tar zxvf unixODBC-2.3.1.tar.gz
$ cd unixODBC-2.3.1
$ ./configure --prefix=/usr --sysconfdir=/etc
$ make
$ make install
```



If you are on a 64-bit system, you have to specify the 64-bit version of libraries with `--libdir` as follows:

```
./configure --prefix=/usr --sysconfdir=/etc --libdir=/usr/lib64
```

The default locations are: `/usr/bin` for binary, and `/usr/lib` or `/usr/lib64` for libraries depending on the version you installed.

Installing database drivers

UnixODBC supports a quite wide and almost complete list of databases. Most of the following widely diffused databases are supported:

- MySQL
- PostgreSQL
- Oracle
- DB2
- Sybase
- Microsoft SQL Server (via FreeTDS)



The complete list of databases supported by UnixODBC is available at <http://www.unixodbc.org/drivers.html>.

MySQL ODBC drivers

Now if you previously installed the UnixODBC via the package manager, you can follow the same procedure, for example, on RedHat with the following command:

```
$ yum install mysql-connector-odbc
```

Otherwise, they are available also as a packet; you only need to download the package, for example, `mysql-connector-odbc-5.1.12-linux-glibc2.5-x86-64bit.tar.gz`.

Then decompress the package and copy the contents in the `/usr/lib/odbc` and `/usr/lib64/odbc/` directories as follows:

```
$ tar xzf mysql-connector-odbc-5.1.12-linux-glibc2.5-x86-64bit.tar.gz
$ mkdir /usr/lib64/odbc/
$ cp /usr/src/mysql-connector-odbc-5.1.12-linux-glibc2.5-x86-64bit/lib/* /usr/lib64/odbc/
```

Now you can check if all the needed libraries are present on your system by using the `ldd` command.

This can be done on a 32-bit system with the following command:

```
$ ldd /usr/lib/odbc/libmyodbc5.so
```

The same can be done on a 64-bit system by using the following command:

```
$ ldd /usr/lib64/odbc/libmyodbc5.so
```

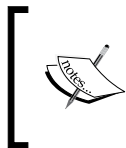
If nothing is marked as `Not Found`, this means that all the needed libraries are found and you can go ahead; otherwise, you need to check what is missing and fix it.

All the installed ODBC database drivers are listed in `odbcinst.ini`; this file, for MySQL 5, should contain the following:

```
[mysql]
Description = ODBC for MySQL
Driver      = /usr/lib/odbc/libmyodbc5.so
```

And the following for a 64-bit system:

```
[mysql]
Description = ODBC for MySQL
Driver      = /usr/lib64/odbc/libmyodbc5.so
```



For all the available ODBC options, refer to the official documentation available at <http://dev.mysql.com/doc/refman/5.0/en/connector-odbc-configuration-connection-parameters.html>.

Data sources are defined in the `odnc.ini` file. You need to create a file with the following content:

```
[mysql-test]
# This is the driver name as specified on odbcinst.ini
Driver = MySQL5
Description = Connector ODBC MySQL5
Database = <db-name-here>
USER= <user-name-here>
Password = <database_password-here>
SERVER = <ip-address-here>
PORT = 3306
```



It is possible to configure ODBC to use a secure connection SSL, but you need to generate a certificate and configure both the sides (ODBC and server) to enable that. Refer to the official documentation for this.

PostgreSQL ODBC drivers

In order to access a PostgreSQL database via ODBC, you need to install the appropriate drivers. They will be used by the Zabbix server, to send the queries to any PostgreSQL database via the ODBC protocol.

The official ODBC drivers for PostgreSQL are available at <http://psqlodbc.projects.pgfoundry.org/>.


Perform the following steps to work with the PostgreSQL database:

1. You can download, compile, and install the psqlODBC driver with the following commands:

```
$ tar -zxvf psqlodbc-xx.xx.xxxx.tar.gz
$ cd psqlodbc-xx.xx.xxxx
$ ./configure
$ make
$ make install
```
2. The configuring script accepts different options; some of the most important ones are as follows:

```
--with-libpq=DIR postgresql path
--with-unixodbc=DIR path or direct odbc_config file (default:yes)
--enable-pthreads= thread-safe driver when available (not on all platforms)
```
3. Once compiled and installed, you can create a template file with the following content:

```
[PostgreSQL]
Description      = PostgreSQL driver for Linux
Driver           = /usr/local/lib/libodbcpsql.so
Setup            = /usr/local/lib/libodbcpsqlS.so
```
4. Now, `odbcinst` can be invoked by passing your template to that command.

 `$ odbcinst -i -d -f template_file` psqlODBC supports encrypted logins with md5, but not with crypt. Bear in mind that *only* the login is encrypted after login. ODBC sends all the queries in plain text. As of Version 08.01.002, psqlODBC supports SSL encrypted connections, which will protect your data.

- As the `psqlODBC` driver supports threads, you can alter the thread serialization level for each driver entry. So for instance, the content of `odbcinst.ini` will be as follows:

```
[PostgreSQL]
Description      = PostgreSQL driver for Linux
Driver           = /usr/local/lib/libodbcpsql.so
Setup           = /usr/local/lib/libodbcpsqlS.so
Threading        = 2
```

- Now you need to configure the `odbc.ini` file. You can also use `odbcinst` here, providing a template or simply a text editor, as follows.

```
$ odbcinst -i -s -f template_file
```

- You should have inside your `odbc.ini` file something like the following:

```
[PostgreSQL]
Description      = Postgres to test
Driver           = /usr/local/lib/libodbcpsql.so
Trace           = Yes
TraceFile        = sql.log
Database         = <database-name-here>
Servername       = <server-name-or-ip-here>
Username         = <username>
Password         = <password>
Port            = 5432
Protocol         = 6.4
ReadOnly         = No
RowVersioning    = No
ShowSystemTables = No
ShowOidColumn    = No
FakeOidIndex     = No
ConnSettings     =
```

Oracle ODBC drivers

Oracle is another database widely used and provides an ODBC driver as well. The following is a description on how to install Oracle's ODBC, because at <http://www.unixodbc.org>, there isn't much information about it.

- The first thing to do is get the instant client from the Oracle website. Oracle provides some of the instant client packets as `rpm` and `tar.gz`, as shown in the following commands:

```
$ rpm -I oracle-instantclient11.2-basic-11.2.0.1.0-1.i386.
rpm oracle-instantclient11.2-odbc-11.2.0.1.0-1.i386.rpm oracle-
instantclient11.2-sqlplus-11.2.0.1.0-1.i386.rpm
```


2. Then you need to configure some environment variables as follows:

```
$ export ORACLE_HOME=/usr/lib/oracle/11.2/client
$ export ORACLE_HOME_LISTNER=/usr/lib/oracle/11.2/client/bin
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH :/usr/lib/oracle/11.2/
client/lib
$ export SQLPATH=/usr/lib/oracle/11.2/client/lib
$ export TNS_ADMIN=/usr/lib/oracle/11.2/client/bin
```

3. Now you need to configure the `odbcinst.ini` file. This file should have the following content:

```
[Oracle11g]
Description = Oracle ODBC driver for Oracle 11g
Driver      =
            /usr/lib/oracle/11.2/client/lib/libsqora.so.11.1
```

4. In the `odbc.ini` file, the relative DSN entry needs to be configured as follows:

```
[ORCLTEST]
Driver      = Oracle 11g ODBC driver
ServerName  = <enter-ip-address-here>
Database    = <enter-sid-here>
DSN         = ORCLTEST
Port        = 1521
```

5. You can test the connection as usual with the following command:

```
$ isql -v ORCLTEST
+-----+
| Connected!                                |
|                                           |
| sql-statement                            |
| help [tablename]                        |
| quit                                    |
+-----+
```

6. Now your ODBC connection is fine.

UnixODBC configuration files

Now you are enabled to connect to most of the common databases. To check the connection, you can test it using `isql`, as follows:

1. If you didn't specify the username and password inside the `odbc.ini` file, it can be passed to the DSN with the following syntax:

```
$ isql <DSN> <user> <password>
```

2. Otherwise, if everything is specified, you can simply check the connection with the following command:

```
$ isql mysql-test
```

3. If all goes well, you should see the following output:

```
+-----+
| Connected!                                |
|                                           |
| sql-statement                            |
| help [tablename]                         |
| quit                                     |
|                                           |
+-----+
SQL>
```



If you get the following error from UnixODBC: **Data source name not found and no default driver specified**, make sure that the `ODBCINI` and `ODBCSYSINI` environment variables are pointing to the right `odbc.ini` file. For example, if your `odbc.ini` file is in `/usr/local/etc`, the environments should be set as follows:

```
export ODBCINI=/usr/local/etc/odbc.ini
export ODBCSYSINI=/usr/local/etc
```

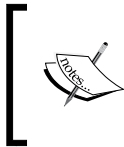
4. If a DSN is presenting issues, the following command can be useful:

```
$ isql -v <DSN>
```

This enables the **verbose mode**, which is very useful to debug a connection.

Compiling Zabbix with ODBC

Now if you connect to the target database that is to be monitored, it is time to compile the Zabbix server with the ODBC support, by performing the following steps.



If your Zabbix is already up and running, don't run the `make install` command during a normal installation as it will copy too many files, and it's possible that some of them will be overwritten. In this case, it is better to just copy the Zabbix server's executable file.

1. Now you can get the `configure` command line with all the options used as specified in *Chapter 1, Deploying Zabbix*, by adding the `--with-unixodbc` parameter as follows:

```
$ ./configure --enable-server --with-postgresql --with-net-snmp
--with-libcurl --enable-ipv6 --with-openipmi --enable-agent
--with-unixodbc
```

2. You should see the following between the output lines:

```
checking for odbc_config... /usr/local/bin/odbc_config
checking for main in -lodbc... yes
checking whether unixodbc is usable... yes
```

3. This will confirm that all the needed ODBC binaries are found and are usable. Once the configuring phase is completed, you can run the following command:

```
$ make
```

4. Once completed, just take a backup of the previous `zabbix_server` file that was installed, and copy the new version.
5. On starting `zabbix_server`, take a look into the logfile, and you should see the following output:

```
***** Enabled features *****
SNMP monitoring: YES
IPMI monitoring: YES
WEB monitoring: YES
Jabber notifications: YES
Ez Texting notifications: YES
ODBC: YES
SSH2 support: YES
IPv6 support: YES
*****
```

This means that all went well.

Database monitor items

Now it is time to use the Zabbix ODBC functionality. In order to do so, you need to create an item of the **Database monitor** type, as shown in the following screenshot:

The screenshot shows the 'Item' configuration form in Zabbix. The form is titled 'Item' and contains the following fields and options:

- Host:** odbc-test-host (with a 'Select' button)
- Name:** web user connected
- Type:** Database monitor (selected from a dropdown)
- Key:** db.odbc.select[<unique short description>] (with a 'Select' button)
- Additional parameters:** A text area containing:


```
DSN=<database source name>
user=<user name>
password=<password>
sql=<query>
```
- Type of information:** Numeric (unsigned) (selected from a dropdown)
- Data type:** Decimal (selected from a dropdown)
- Units:** (empty text field)
- Use custom multiplier:** ☐ (checked) with a value of 1
- Update interval (in sec):** 30
- Flexible intervals:** A table with columns 'Interval', 'Period', and 'Action'. The table is currently empty, with the text 'No flexible intervals defined.' below the headers.
- New flexible interval:** A section at the bottom with fields for 'Interval (in sec)' (50) and 'Period' (1-7,00:00-24:00), followed by an 'Add' button.

The item where the retrieved value will be stored is identified by the item key as follows:

```
db.odbc.select[<unique short description>]
```

The <unique short description> is a string that must be unique, and can be whatever you want. An example is as follows:

```
db.odbc.select[web_user_connected_on_myapp]
```

Inside the **Additional parameters** field, you need to specify the following:

```
DSN=<database source name>
user=<user name>
password=<password>
sql=<query>
```

Where the DSN should exist in `/etc/odbc.ini`, and whether the username and password are stored in the DSN definition or not, can be specified here. In the last line, you need to specify the SQL query.

Some considerations about the ODBC SQL query

The following are some restrictions and things to consider on a SQL query:

- The SQL must begin with a select clause
- The SQL can't contain any line breaks
- The query must return only one value
- If the query returns multiple columns, only the first one is read
- If the query returns multiple rows, only the first column of the first row is read
- Macros are not be replaced (for example, {HOSTNAME})
- The SQL command must begin with lowercase, that is, `sql=`
- The query needs to terminate before the timeout
- The query must exactly return the value type specified, otherwise the item will be unsupported

As you can see, there are only some limitations that you can accept. In particular, you can't call a function if that one function returns only one value. You can't execute a stored procedure; you can only select the data. Also, the query can't contain any line breaks, so long and complex queries will not be easily readable.

The following are some other points to consider:

- If the database is particularly loaded, it can respond with a delay (the login can also suffer a delay caused by the workload)
- Every query executes a log in
- If you use proxies, they too need to be compiled with the UnixODBC support

If you consider to have a database that will be under heavy stress, don't have a pool introduced for an overhead that is not necessary. Also, in this case, it is possible that just to have a connection, you need to wait for more than five seconds.

The five seconds previously mentioned is not a random value; indeed, the timeout of a connection is defined when you open a connection. During the initialization of that, you need to define your expected timeout before considering the connection impossible.

Zabbix defines this timeout in the following command:

```
src/libs/zbxdbhigh/odbc.c
```

On line 130 of that file, we have the definition of the connection timeout for Zabbix as follows:

```
SQLSetConnectAttr(pdbh->hdbc, (SQLINTEGER)SQL_LOGIN_TIMEOUT,
    (SQLPOINTER)5, (SQLINTEGER)0);
```

This (SQLPOINTER) 5 sets SQL_LOGIN_TIMEOUT to five seconds. If your database doesn't respond in five seconds, you will get the following error inside the logfile:

```
[ODBC 3.51 Driver]Can't connect to MySQL server on 'XXX.XXX.XXX.XXX' (4)
(2003).
```



In case of SQL_LOGIN_TIMEOUT, you can consider increasing it to 15 seconds and recompile the server and proxy as follows:

```
SQLSetConnectAttr(pdbh->hdbc,
    (SQLINTEGER)SQL_LOGIN_TIMEOUT, (SQLPOINTER)15,
    (SQLINTEGER)0);
```

Zabbix JMX monitoring

Version 2.0 of Zabbix has a native support to monitor applications using JMX. The actor that monitors the JMX application is a Java daemon called **Zabbix Java gateway**. Basically, it works like a gateway. When Zabbix wishes to know the value of some JMX counter, it simply asks the Java gateway, and the gateway will do all the work for Zabbix. All the queries are done using the JMX management API from Oracle.

The Zabbix Java gateway is in early development, providing great functionality but still experiencing some challenges.

The point of force of this method is that the application only needs to be started with the JMX remote console enabled, and doesn't need to implement or extend the class or write a new code to handle the Zabbix request because the entire request is a standard JMX.

The default way to enable the JMX console is to start the Java application with the following parameters:

```
-Dcom.sun.management.jmxremote  
-Dcom.sun.management.jmxremote.port=<put-your-port-number-here>  
-Dcom.sun.management.jmxremote.authenticate=false  
-Dcom.sun.management.jmxremote.ssl=false
```

With these parameters, you are going to configure the JMX interface on the application's side. As usual, you need to define a port, the authentication method, and the encryption.

This basic setup is the simplest and easiest way, but unfortunately, it is not the safest and most secure configuration.

Considering some JMX security aspects

Now, as you are going to open a door in your application, you are basically exposing your application to a security attack. The JMX console, on most of the widely diffused application servers, is not only an entry point to get values from the counter, but also something that is a lot more sophisticated. Basically, with a JMX console open in an application server, you can deploy an application. Start it, stop it, and so on, as you can figure out what a hacker can deploy, and start an application, or cause an issue on the running one. The JMX console can be called from the application server looping itself, using the `post` and `get` methods. Adding malicious content in the `HEAD` section of a web page results in the server that has a JMX console not secured easily hackable, which is the weakest point of your infrastructure. Once an application server is compromised, your entire network is potentially exposed, and you need to prevent all this. This can be done through the following steps:

1. The first thing to do is enable the authentication as follows:

```
-Dcom.sun.management.jmxremote.authenticate=true
```

2. Now you need to specify a file that will contain your password as follows:

```
-Dcom.sun.management.jmxremote.password.file=/etc/java-6-  
penjdk/management/jmxremote.password
```



There are potential security issues with password authentication for JMX remote connectors. Once the client obtains the remote connector from an insecure RMI registry (the default), like for all the man-in-the-middle attacks, an attacker can start a bogus RMI registry on the target server, right before the valid original one is started, and can then steal the client's passwords.

3. Another good thing to do is to profile the users, specifying the following parameter:


```
-Dcom.sun.management.jmxremote.access.file=/etc/java-6-  
penjdk/management/jmxremote.access
```
4. The access file, for instance, should contain something like the following:


```
monitorRole readonly  
controlRole readwrite
```
5. The password file should be as follows:


```
monitorRole <monitor-password-here>  
controlRole <control-password-here>
```
6. Now to avoid password stealing, you should enable the SSL as follows:


```
-Dcom.sun.management.jmxremote.ssl=true
```
7. This parameter is consequently tied with the following ones:


```
-Djavax.net.ssl.keyStore=<Keystore-location-here>  
-Djavax.net.ssl.keyStorePassword=<Default-keystore-  
password>  
-Djavax.net.ssl.trustStore=<Truststore-location-here>  
-Djavax.net.ssl.trustStorePassword=<Truststore-password-here>  
-Dcom.sun.management.jmxremote.ssl.need.client.auth=true
```



The -D parameter will be written in the startup file of your application or application server, as after this configuration, your startup file will contain sensitive data (your keystore and truststore passwords) that needs to be protected and not readable from other accounts in the same group, or other users.

Installing a Zabbix Java gateway

To compile the Java gateway, perform the following steps:

1. First, you need to run the following:


```
$ ./configure --enable-java
```
2. You should get an output as follows:


```
Enable Java gateway:   yes  
Java gateway details:  
  Java compiler:       javac  
  Java archiver:       jar
```


3. This shows that the Java gateway is going to be enabled and compiled after the following command is used:

```
$ make && make install
```

4. The Zabbix Java gateway will be installed at the following location:

```
$PREFIX/sbin/zabbix_java
```

5. Basically, the directory structure will contain the following file—the java gateway:

```
bin/zabbix-java-gateway-2.0.5.jar
```

6. The libraries needed by the gateway are as follows:

```
lib/logback-classic-0.9.27.jar
```

```
lib/logback-console.xml
```

```
lib/logback-core-0.9.27.jar
```

```
lib/logback.xml
```

```
lib/org-json-2010-12-28.jar
```

```
lib/slf4j-api-1.6.1.jar
```

7. The scripts to start and stop the gateway are as follows:

```
shutdown.sh
```

```
startup.sh
```

8. This is a common script sourced from the start and stop scripts that contain the following configuration:

```
settings.sh
```

9. Now, if you have enabled the SSL communication, you need to enable the same security level on the Zabbix Java gateway. To do this, you need to add the following parameter in the startup script:

```
-Djavax.net.ssl.*
```

10. Once all this is set, you need to specify the following inside the Zabbix server configuration:

```
JavaGateway=<ip-address-here>
```

```
JavaGatewayPort=10052
```



If you would like to use the Java gateway from your proxy, you need to configure both `JavaGateway` and `JavaGatewayProperties` in the proxy configuration file.

11. Since by default, Zabbix doesn't start any Java poller, you need to specify that also, as follows:

```
StartJavaPollers=5
```

12. Then, restart the Zabbix server or proxy once done.
13. Now you can finally start the Zabbix Java gateway by running the `startup.sh` command.

The logs will be available at `/tmp/zabbix_java.log` with verbosity as "info".



As the Zabbix Java gateway uses the logback library, you can change the log level or the logfile location by simply changing the `lib/logback.xml` file. In particular, the following XML tags need to be changed:

```
<fileNamePattern>/tmp/zabbix_java.log.%i</
fileNamePattern><root level="info">
```

Here you can change all the logrotation parameters as well.

Configuring Zabbix JMX

Now it is time to create a JMX monitored host, with its relatively monitored JMX items. To do that, inside the host configuration, you need to add a JMX interface and address as shown in the following screenshot:

Agent interfaces	IP address	DNS name	Connect to	Port	Default
	Add				
SNMP interfaces	Add				
JMX interfaces	↓ 192.168.1.1		IP DNS	12345	⊙ Remove
	Add				
IPMI interfaces	Add				
Monitored by proxy	(no proxy) ▼				
Status	Monitored ▼				

Once you have done that for each of the JMX counters you want to acquire, you need to define an item of the **JMX agent** type. Inside the definition of the JMX agent, you need to specify the username and password, and the JMX query string. The JMX key is composed of the following:

- Object name of MBean
- Attribute name, that is, the attribute name of MBean

The following screenshot shows the **Item** configuration window:

Item

Hostjmx-test-hostSelect

Name

TypeJMX agent

Keyjmx[<object name>,<attribute name>]Select

Host interface192.168.1.1 : 12345

User name

Password

Type of informationNumeric (unsigned)

Data typeDecimal

Units

Use custom multiplier☐1

Update interval (in sec)30

Flexible intervals

Interval	Period	Action
No flexible intervals defined.		

New flexible interval

Interval (in sec)50

Period1-7,00:00-24:00

Add

Keep history (in days)90

Keep trends (in days)365

Store valueAs is

Show valueAs is[show value mappings](#)

New application

Applications

-None-

Populates host inventory field-None-

Description

StatusEnabled

Save

Cancel

The field **Data type** in this configuration window permits to store the unsigned integer values (such as 0 and 1) as numbers, or as Boolean values (such as true or false).

JMX keys in detail

MBean is a quite simple string defined in your Java application. The other component is a bit more complex indeed; the attribute can return primitive data types or composite data.

The primitive data types are simple types such as integer and strings. For instance, you can have a query such as the following:

```
jmx[com.example:Type=Test,weight]
```

This will return the weight expressed as a numerical floating point value.

If the attribute returns composite data, it is a bit more complicated, but is handled since dots are supported. For instance, you can have a pen that can have two values that represent color and the remaining ink, usually dot separated, as shown in the following code:

```
jmx[com.example:Type=Test,pen.remainink]
jmx[com.example:Type=Test,pen.color]
```

Now if you have an attribute name that includes a dot in its name, such as `all.pen`, you need to escape the dot, as shown in the following code:

```
jmx[com.example:Type=Test,all\.pen.color]
```

If your attribute name also contains a backslash (`\`), this needs to be escaped twice, as shown in the following code:

```
jmx[com.example:Type=Test,c:\\utility]
```

If the object name or attribute name contains spaces or commas, it needs to be double-quoted:

```
jmx[com.example:type=Hello,"c:\\documents and settings"]
```

Issues and considerations about JMX

Unfortunately, JMX support is not as flexible and customizable as it should be; at least at the time of writing this book, JMX still had some problems.

For instance, from my personal experience, I know that JBOSS, which is one of the most widely used application servers, can't be successfully enquired. The JMX endpoint is currently hardcoded into `JMXItemChecker.java` as follows:

```
service:jmx:rmi:///jndi/rmi://" + conn + ":" + port + "/jmxrmi"
```

Some applications use different endpoints for their JMX management console. JBOSS is one of them. The endpoint is not configurable as per the host or frontend, and you can't add a parameter to specify this endpoint on the host's configuration window.

Anyway, the development is really active and things are getting better and are improving everyday. At the moment, the status is that the Zabbix Java gateway needs some improvement. Also, the current implementation of Zabbix Java gateway suffers because of the workload; if you have more than 100 JMX items per host, the gateway needs to be restarted periodically. It is possible that you face some errors inside your logfile of the following kind:

```
failed: another network error, wait for 15 seconds
```

Followed by:

```
connection restored
```

It is fundamental that the implementer of Zabbix monitoring infrastructure knows not only all the strong points of the product but also the cons and limitations. The implementer can then choose if they want to develop something in-house, use an open source alternative, try to fix the possible issues, or ask the Zabbix team for a new functionality or fix.

Zabbix SNMP monitoring

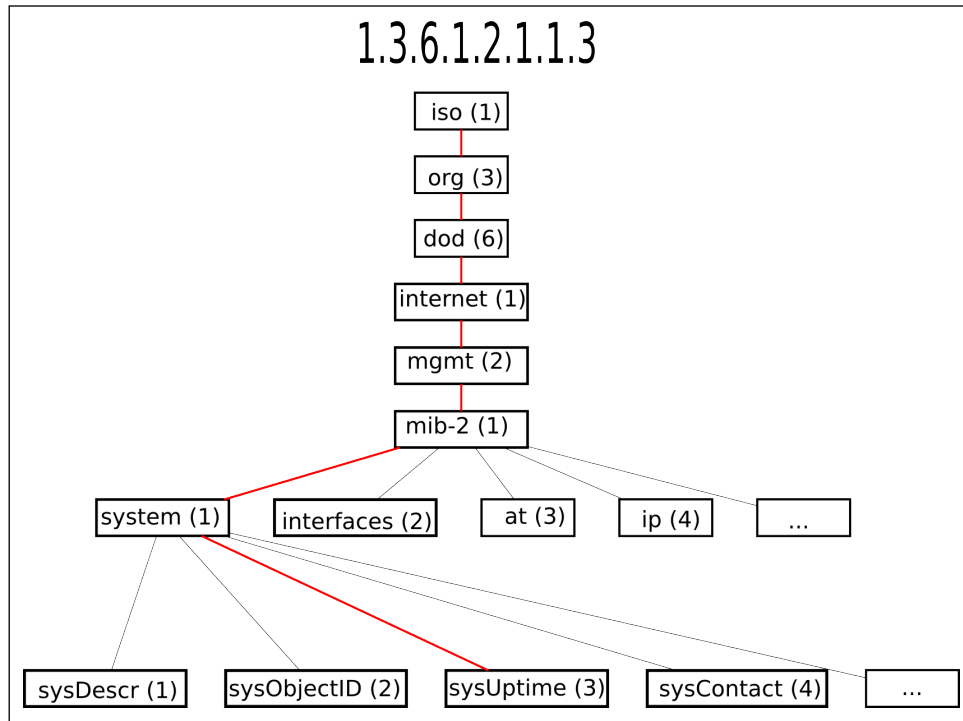
Simple Network Monitoring Protocol (SNMP) may not be as simple as the name suggests; it's a de facto standard for many appliances and applications. It's not just ubiquitous, it's often the only sensible way one can extract the monitoring information from a network switch, a disk array enclosure, a UPS battery, and so on.

The basic architecture layout for SNMP monitoring is actually straightforward. Every monitored host or appliance runs an SNMP agent. This agent can be queried by any probe (whether it's just a command-line program to do manual queries, or a monitoring server like Zabbix) and will send back information on any metric it has made available, or even change some predefined settings on the host itself, as a response to a `set` command from the probe. Furthermore, the agent is not just a passive entity that responds to the `get` and `set` commands, but can also send warnings and alarms as SNMP traps to a predefined host when some specific conditions arise.

Things get a little more complicated when it comes to metric definitions. Unlike a regular Zabbix item, or any other monitoring system, an SNMP metric is part of a huge hierarchy, a tree of metrics that spans hardware vendors and software implementers across all of the IT landscape. This means that every metric has to be uniquely identified with some kind of code. This unique metric identifier is called **OID**, and identifies both the object and its position in the SNMP hierarchy tree.

OIDs and their values are the actual content that is passed in the SNMP messages. While this is most efficient from a network traffic point of view, OIDs need to be translated into something usable and understandable by humans as well. This is done by using a distributed database called **Management Information Base (MIB)**. MIBs are essentially text files that describe a specific branch of the OID tree, with a textual description of its OIDs, their data types, and a human-readable string identifier.

MIBs let us know, for example, that OID **1.3.6.1.2.1.1.3** refers to the system-uptime of whatever machine the agent is running on. Its value is expressed as an integer, in hundredth of a second, and can generally be referred to as `sysUptime`.



As you can see, this is quite different from the way the Zabbix agent items work, both in terms of the connection protocol, and item definition and organization. Nevertheless, Zabbix provides facilities to translate from SNMP OIDs to Zabbix items – if you compiled the support for the server in SNMP, it will be able to create the SNMP queries natively, and with the help of a couple of supporting tools, it will also be able to process SNMP traps.

This is of course an essential feature if you need to monitor appliances that only support SNMP, and have no way of installing a native agent on network appliances in general (switcher, routers, and so forth), disk array enclosures, and so on. But the following may be some reasons for you to actually choose SNMP as the main monitoring protocol in your network and completely dispense of Zabbix agents:

- You may not need many complex or custom metrics apart from what is already provided by an operating system's SNMP OID branch. You most probably have already set up SNMP monitoring for your network equipment, and if you just need simple metrics such as uptime, CPU load, free memory, and so on, from your average host, it might be simpler to rely on SNMP for it as well, instead of the native Zabbix agent. This way, you will never have to worry about agent deployment and updates – you just let the Zabbix server contact the remote SNMP agents and get the information you need.
- The SNMP protocol and port numbers are well-known by virtually all the products. If you need to send monitoring information across networks, it might be easier to rely on the SNMP protocol instead of the Zabbix one. This could be because traffic on the UDP ports 161 and 162 is already permitted, or because it might be easier to ask a security administrator to allow access to a well-known protocol instead of a relatively more obscure one.
- SNMP Version 3 features built-in authentication and security. This means that, contrary to the Zabbix protocol, as you have already seen in *Chapter 2, Distributed Monitoring*, SNMPv3 messages will have integrity, confidentiality, and authentication. While Zabbix does support all three versions of SNMP, it's strongly advised that you use Version 3 wherever possible, because it's the only one with real security features. In contrast, Versions 1 and 2 only have a simple string sent inside a message as a very thin layer of security.
- While there may be good reasons to use SNMP monitoring as much as possible in your Zabbix installation, there are still a couple of strong ones to stick with the Zabbix agent. The Zabbix agent has a few, very useful built-in metrics that would need custom extensions if implemented through an SNMP agent. For example, if you want to monitor a logfile, with automatic log rotation support, and skipping old data, you just need to specify the `logrt []` key for a Zabbix active item. The same goes if you want to monitor the checksum, or the size of a specific file, the Performance Monitor facility of the Windows operating system, and so on. In all these cases, the Zabbix agent is the most immediate and simple choice.

- The Zabbix agent has the ability to discover many kinds of resources that are available on the host, and report them back to the server, which will in turn automatically create items and triggers, and destroy them when the said resources are not available anymore. This means that with the Zabbix agent, you will be able to let the server create the appropriate items for every host's CPU, mounted filesystem, number of network interfaces, and so on. While it's possible to define low-level discovery rules based on SNMP, it's often easier to rely on the Zabbix agent for this kind of functionality.

So once again, you have to balance the different features of each solution in order to find the best match for your environment. But generally speaking, you could make the following broad assessments: if you have simple metrics but need strong security, go with SNMP v3; if you have complex monitoring or automated discovery needs and can dispense with strong security (or are willing to work harder to get it, as explained in *Chapter 2, Distributed Monitoring*), go with the Zabbix agent and protocol.

That said, there are a couple of aspects worth exploring when it comes to Zabbix SNMP monitoring. We'll first talk about simple SNMP queries, and then about SNMP traps.

SNMP queries

An SNMP monitoring item is quite simple to configure. The main point of interest is that while the server will use the OID you provided to get the measurement, you'll still need to define a unique name for the item, and most importantly, a unique item key. Keep in mind that an item key is used in all of Zabbix's expressions that define triggers, calculated items, actions, and so on. So try to keep it short and simple, while easily recognizable. As an example, suppose you want to define a metric for the incoming traffic on network port number 3 of an appliance, the OID would be **1.3.6.1.2.1.2.2.1.10.3**, while you could call the key something like **port3.ifInOctects**, as shown in the following screenshot:

Item

Host

Template App Agentless

Select

Name

Incoming traffic on port 3

Type

SNMPv3 agent ▼

Key

port3.ifInOctets

Select

SNMP OID

.1.3.6.1.2.1.2.2.1.10.3

SNMPv3 security

name

SNMPv3 security

authPriv ▼

level

SNMPv3 auth

passphrase

SNMPv3 priv

passphrase

Port

161

Type of information

Numeric (unsigned) ▼

Data type

Decimal ▼

Units

Use custom multiplier

☐

1

Update interval (in sec)

30

Flexible intervals

Interval	Period	Action
No flexible intervals defined.		

New flexible interval

Interval (in sec)

50

Period

1-7,00:00-24:00

Add

Keep history (in days)

90

Keep trends (in days)

365

Store value

Delta (speed per second) ▼

If you don't already have your SNMP items defined in a template, an easy way to get them is by using the `snmpwalk` tool to directly query the host you need to monitor, and get information about the available OIDs and their data types.

For example, the following command is used to get the whole object tree from the appliance at 10.10.15.19:

```
$ snmpwalk -v 3 -l AuthPriv -u user -a MD5 -A auth -x DES -X priv -m ALL 10.10.15.19
```



You need to substitute the string `user` with the username for the SNMP agent, `auth` with the authentication password for the user, `priv` with the privacy password, `MD5` with the appropriate authentication protocol, and `DES` with the privacy protocol you defined for the agent.

The SNMP agent on the host will respond with a list of all its OIDs. The following is a fragment of what you could get:

```
HOST-RESOURCES-MIB::hrSystemUptime.0 = Timeticks: (8609925) 23:54:59.25

HOST-RESOURCES-MIB::hrSystemDate.0 = STRING: 2013-7-28,9:38:51.0,+2:0

HOST-RESOURCES-MIB::hrSystemInitialLoadDevice.0 = INTEGER: 393216

HOST-RESOURCES-MIB::hrSystemInitialLoadParameters.0 = STRING: "root=/dev/
sda8 ro"

HOST-RESOURCES-MIB::hrSystemNumUsers.0 = Gauge32: 2

HOST-RESOURCES-MIB::hrSystemProcesses.0 = Gauge32: 172

HOST-RESOURCES-MIB::hrSystemMaxProcesses.0 = INTEGER: 0

HOST-RESOURCES-MIB::hrMemorySize.0 = INTEGER: 8058172 KBytes

HOST-RESOURCES-MIB::hrStorageDescr.1 = STRING: Physical memory

HOST-RESOURCES-MIB::hrStorageDescr.3 = STRING: Virtual memory

HOST-RESOURCES-MIB::hrStorageDescr.6 = STRING: Memory buffers

HOST-RESOURCES-MIB::hrStorageDescr.7 = STRING: Cached memory

HOST-RESOURCES-MIB::hrStorageDescr.8 = STRING: Shared memory
```

```
HOST-RESOURCES-MIB::hrStorageDescr.10 = STRING: Swap space
```

```
HOST-RESOURCES-MIB::hrStorageDescr.35 = STRING: /run
```

```
HOST-RESOURCES-MIB::hrStorageDescr.37 = STRING: /dev/shm
```

```
HOST-RESOURCES-MIB::hrStorageDescr.39 = STRING: /sys/fs/cgroup
```

```
HOST-RESOURCES-MIB::hrStorageDescr.53 = STRING: /tmp
```

```
HOST-RESOURCES-MIB::hrStorageDescr.56 = STRING: /boot
```

Let's say that we are interested in the system's memory size. To get the full OID for it, we will reissue the `snmpwalk` command using the `fn` option for the `-o` switch. These will tell `snmpwalk` to display the full OIDs in a numeric format. We will also limit the query to the OID we need, as taken from the previous output:

```
$ snmpwalk -v 3 -l AuthPriv -u user -a MD5 -A auth -x DES -X priv -m ALL
-O fn 10.10.15.19 HOST-RESOURCES-MIB::hrMemorySize.0
.1.3.6.1.2.1.25.2.2.0 = INTEGER: 8058172 KBytes
```

And there we have it. The OID we need to put in our item definition is
`1.3.6.1.2.1.25.2.2.0`.

SNMP traps

SNMP traps are a bit of an oddball if compared to all the other Zabbix item types. Unlike other items, SNMP traps do not report a simple measurement, but an event of some type. In other words, they are the result of some kind of a check or computation made by the SNMP agent and sent over to the monitoring server as a status report. An SNMP trap can be issued every time a host is rebooted, an interface is down, a disk is damaged, or a UPS has lost power and is keeping the servers up using its battery.

This kind of information contrasts with Zabbix's basic assumption, that is, an item is a simple metric not directly related to a specific event. On the other hand, there may be no other way to be aware of certain situations if not through an SNMP trap, either because there are no related metrics (consider for example, the event "the server is being shut down"), or because the appliance's only way to convey its status is through a bunch of SNMP objects and traps.

So traps are of relatively limited use to Zabbix, as you can't do much more than build a simple trigger out of every trap, and then notify about the event (not much point in graphing a trap, or building calculated items on it). Nevertheless, they may prove essential for a complete monitoring solution.

To manage SNMP traps effectively, Zabbix needs a couple of helper tools: the **snmptrapd** daemon, to actually handle connections from the SNMP agents, and some kind of script to correctly format every trap and pass it to the Zabbix server for further processing.

The snmptrapd process

If you have compiled an SNMP support into the Zabbix server, you should already have the complete SNMP suite installed, which contains the SNMP daemon, the SNMP trap daemon, and a bunch of utilities such as `snmpwalk` and `snmptrap`.

If it turns out that you don't actually have the SNMP suite installed, the following command should take care of the matter:

```
# yum install net-snmp net-snmp-utils
```

Just as the Zabbix server has a bunch of daemon processes that listen on the TCP port 10051 for incoming connections (from agents, proxies, and nodes), `snmptrapd` is the daemon process that listens on the UDP port 162 for incoming traps coming from remote SNMP agents.

Once installed, `snmptrapd` reads its configuration options from an `snmptrapd.conf` file, which can be usually found in the `/etc/snmp/` directory. The bare minimum configuration for `snmptrapd` requires only the definition of a community string in the case of SNMP Versions 1 and 2, which is as follows:

```
authCommunity log public
```

Or the definition of a user and a privacy level in the case of SNMP Version 3, which is as follows:


```
createUser -e ENGINEID user MD5 auth DES priv
```



You need to create a separate `createUser` line for every remote Version 3 agent that will send traps. You also need to substitute all the `user`, `auth`, `priv`, `MD5`, and `DES` strings to what you have already configured on the agent, as explained in the previous note. Most importantly, you need to set the correct `ENGINEID` for every agent. You can get it from the agent's configuration itself.

With this minimal configuration, `snmptrapd` will limit itself to log the trap to `syslog`. While it could be possible to extract this information and send it to Zabbix, it's easier to tell `snmptrapd` how it should handle the traps. While the daemon has no processing capabilities of its own, it can execute any command or application by either using the `trapHandle` directive, or leveraging its embedded `perl` functionality. The latter is more efficient as the daemon won't have to fork a new process and wait for its execution to finish, so it's the recommended one if you plan to receive a significant number of traps. Just add the following line to `snmptrapd.conf`:

```
perl do "/usr/local/bin/zabbix_trap_receiver.pl";
```

 You can get the `zabbix_trap_receiver` script from the Zabbix sources. It's located in `misc/snmptrap/zabbix_trap_receiver.pl`.

Once restarted, the `snmptrapd` daemon will execute the `perl` script of your choice to process every trap received. As you can probably imagine, your job doesn't end here—you still need to define how to handle the traps in your script, and find a way to send the resulting work over to your Zabbix server. We'll discuss both of these aspects in the following section.

The Perl trap handler

The `perl` script included in the Zabbix distribution works as a translator from an SNMP trap format to a Zabbix item measurement. For every trap received, it will format it according to the rules defined in the script, and will output the result in a logfile. The Zabbix server will in turn monitor the said logfile, and process every new line as an SNMP trap item, basically matching the content of the line to any trap item defined for the relevant host. Let's see how it all works by looking at the `perl` script itself and illustrating its logic:

```
#!/usr/bin/perl

#
# Zabbix
# Copyright (C) 2001-2013 Zabbix SIA
#
# This program is free software; you can redistribute it and/or
# modify
# it under the terms of the GNU General Public License as
# published by
# the Free Software Foundation; either version 2 of the License,
```

```
# or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public
# License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
# 02110-1301, USA.
#

#####
#### ABOUT ZABBIX SNMP TRAP RECEIVER ####
#####

# This is an embedded perl SNMP trapper receiver designed for
# sending data to the server.

# The receiver will pass the received SNMP traps to Zabbix server
# or proxy running on the
# same machine. Please configure the server/proxy accordingly.
#
# Read more about using embedded perl with Net-SNMP:
#   http://net-snmp.sourceforge.net/wiki/index.php/Tut:Extending_
# snmpd_using_perl
```

This first section contains just the licensing information and a brief description of the script. Nothing worth mentioning, except a simple reminder – check that your perl executable is correctly referenced in the first line, or change it accordingly. The following section is more interesting, and if you are happy with the script's default formatting of SNMP traps, it may also be the only section you will ever need to customize:

```
#####
#### ZABBIX SNMP TRAP RECEIVER CONFIGURATION ####
#####

### Option: SNMPTrapperFile
#   Temporary file used for passing data to the server (or
#   proxy). Must be the same
```

```

#      as in the server (or proxy) configuration file.
#
# Mandatory: yes
# Default:
$SNMPTrapperFile = '/tmp/zabbix_traps.tmp';

### Option: DateTimeFormat
#      The date time format in strftime() format. Please make
#      sure to have a corresponding
#      log time format for the SNMP trap items.
#
# Mandatory: yes
# Default:
$DateTimeFormat = '%H:%M:%S %Y/%m/%d';

```

Just set `$SNMPTrapperFile` to the path of the file you wish the script to log its trap to, and set the `SNMPTrapperFile` option in your `zabbix_server.conf` file to the same value. While you are at it, also set `StartSNMPTrapper` to 1 in `zabbix_server.conf` so that the server will start monitoring the said file.

The `$DateTimeFormat`, on the other hand, should match the format of the actual SNMP traps you receive from the remote agents. Most of the time, the default value is correct, but take the time to check it and change it as needed.

The following section contains the actual logic of the script. Notice how the bulk of the logic is contained in a subroutine called `zabbix_receiver`. This subroutine will be called and executed towards the end of the script, but is worth examining in detail.

```

#####
#### ZABBIX SNMP TRAP RECEIVER ####
#####
use Fcntl qw(O_WRONLY O_APPEND O_CREAT);
use POSIX qw(strftime);

sub zabbix_receiver
{
    my (%pdu_info) = %{$_[0]};
    my (@varbinds) = @{$_[1]};

```


The `snmptrapd` daemon will execute the script and pass the trap that it just received. The script will in turn call its subroutine, which will immediately distribute the trap information in two lists – the first argument is assigned to the `%pdu_info` hash and the second one to the `@varbinds` array.

```
# open the output file
unless (sysopen(OUTPUT_FILE, $SNMPTrapperFile,
  O_WRONLY|O_APPEND|O_CREAT, 0666))
{
    print STDERR "Cannot open [$SNMPTrapperFile]:
    $!\n";
    return NETSNMPTRAPD_HANDLER_FAIL;
}
```

Here, the script will open the output file, or fail gracefully if it somehow cannot. The next step consists of extracting the hostname (or IP address) of the agent that sent the trap. This information is stored in the `%pdu_info` hash we defined previously:

```
# get the host name
my $hostname = $pdu_info{'receivedfrom'} || 'unknown';
if ($hostname ne 'unknown') {
    $hostname =~ /\[([.*?)]\].*/;
    $hostname = $1 || 'unknown';
}
```

Now we are ready to build the actual SNMP trap notification message. The first part of the output will be used by Zabbix to recognize the presence of a new trap (by looking for the `ZBXTRAP` string, and knowing which of the monitored hosts the trap refers to). Keep in mind that the IP address or hostname set here must match the SNMP address value in the host configuration as set using the Zabbix frontend. This value must be set even if it's identical to the main IP/hostname for a given host. Once the Zabbix server will have identified the correct host, it will discard this part of the trap notification.

```
# print trap header
#     timestamp must be placed at the beginning of the
#     first line (can be omitted)
#     the first line must include the header "ZBXTRAP
#     [IP/DNS address] "
#         * IP/DNS address is the used to find the
#         corresponding SNMP trap items
#         * this header will be cut during processing
#         (will not appear in the item value)
printf OUTPUT_FILE "%s ZBXTRAP %s\n",
    strftime($DateTimeFormat, localtime), $hostname;
```

After the notification header, the script will output the rest of the trap as received by the SNMP agent.

```
# print the PDU info
print OUTPUT_FILE "PDU INFO:\n";
foreach my $key (keys(%pdu_info))
{
    printf OUTPUT_FILE "  %-30s %s\n", $key,
        $pdu_info{$key};
}
```

The `printf` statement in the previous code will circle over the `%pdu_info` hash and output every key-value pair.

```
# print the variable bindings:
print OUTPUT_FILE "VARBINDS:\n";

foreach my $x (@varbinds)
{
    printf OUTPUT_FILE "  %-30s type=%-2d value=%s\n",
        $x->[0], $x->[2], $x->[1];
}
close (OUTPUT_FILE);
return NETSNMPTRAPD_HANDLER_OK;
}
```

The second `printf` statement, `printf OUTPUT_FILE " %-30s type=%-2d value=%s\n", $x->[0], $x->[2], $x->[1];`, will output the contents of the `@varbinds` array one by one. This array is the one that contains the actual values reported by the trap. Once done, the logfile is closed, and the execution of the subroutine ends with an exit message.

```
NetSNMP::TrapReceiver::register("all", \&zabbix_receiver) or
    die "failed to register Zabbix SNMP trap receiver\n";
print STDOUT "Loaded Zabbix SNMP trap receiver\n";
```

The last few lines of the script set the `zabbix_receiver` subroutine as the actual trap handler, and give some feedback about its correct setup. Once the trap handler will start populating the `zabbix_traps.log` logfile, you need to define the corresponding Zabbix items.

As you've already seen, the first part of the log line is used by the Zabbix trap receiver to match a trap with its corresponding host. The second part is matched to the said host's SNMP trap items regexp definitions, and its content added to every matching item's history of values. This means that if you wish to have a startup trap item for a given host, you'll need to configure an SNMP trap item with an `snmptrap["coldStart"]` key, as shown in the following screenshot:

The screenshot shows the 'Item' configuration form in Zabbix. The 'Host' field is set to 'Template App Agentless'. The 'Name' field contains 'SNMP Agent was restarted'. The 'Type' is set to 'SNMP trap'. The 'Key' field contains 'snmptrap["coldStart"]'. The 'Type of information' is set to 'Text'. The 'Keep history (in days)' field is set to '90'. There is a 'New application' field. The 'Applications' dropdown menu is open, showing '-None-' and 'Services'. The 'Populates host inventory field' dropdown is set to '-None-'. The 'Description' field is empty.

From now on, you'll be able to see the contents of the trap in the item's data history.

Web pages monitoring

In this time and age, web applications are virtually ubiquitous. Some kind of website or a collection of web pages is typically the final product or the service of a complex structure that comprises of different databases, application servers, web servers, proxies, network balancers and appliances, and more. When it comes to monitoring duties, it makes sense to go just a step further and monitor a resulting site or web page in addition to all the backend assets that enable the said page.

The advantages as far as warnings and notifications go, are fairly limited, as failure to reach a web page is certainly a critical event, but it hardly gives any insight on what may be the actual problem if you haven't set up the correct metrics and triggers on the backend side. On the other hand, it may be crucial to have a collection of data about a website's performance, in order to anticipate possible problems, substantiate SLA reporting, and plan for hardware or software upgrades.

One big advantage of Zabbix's web-monitoring facilities is the scenario concept. You can define a single web scenario that is composed of many simple steps, each one building on the previous, and sharing a common set of data. Furthermore, every such definition includes the automatic creation of meaningful items and graphs, both at the scenario level (overall performance) and at the single-step level (local performance). This makes it possible to not only monitor a single web page, but to simulate entire web sessions, so that every component of a web application will contribute to the final results. A single scenario can be very complex and requires a great number of items that would end up being difficult to track and group together. This is the reason why web monitoring in Zabbix has its own web configuration tab and interface, separate from regular items, where you can configure monitoring on a higher level.

Web scenarios support plain HTTP and HTTPS, Basic, NTLM, form-based authentication, cookies, submission of form fields, and checking of page content in addition to the HTTP code responses.

For all their power, web scenarios also suffer a few limitations when it comes to monitoring the modern Web.

First of all, JavaScript is not supported, so you can't simulate a complete AJAX session just like a human user would experience it. This also means that any kind of automated page reloads won't be executed in the scenario.

Furthermore, while you can submit forms, you have to know in advance, both, the name of the fields and their content. If either of them is generated dynamically from page to page (as many ASP.NET pages do, to keep the session information), you won't be able to use it in subsequent steps.

These may look like negligible limitations, but they may prove quite important if you need to monitor any site that relies heavily on client-side elaborations (JavaScript and friends) or on dynamic tokens and form fields. The reality is that an increasing number of web applications or frameworks use one or both of these features.

Nevertheless, even with these limitations, Zabbix's web monitoring facilities prove to be a very useful and powerful tool that you may want to take full advantage of, especially if you produce a lot of web pages as the final result of an IT pipeline.

Authenticating on web pages

The three methods of authentication supported for web monitoring are Basic, NTLM and form based. The first two are fairly straightforward and just need to be defined at the scenario level, as shown in the following screenshot:

The screenshot shows the 'CONFIGURATION OF WEB MONITORING' window with the 'Steps' tab selected. The configuration is as follows:

- Application:** Web Scenarios (with a 'Select' button)
- Name:** Test website
- Authentication:** NTLM authentication (dropdown menu)
- User:** webuser
- Password:** password
- Update interval (in sec):** 60
- Agent:** Internet Explorer 10.0 (dropdown menu)
- Variables:** (empty text area)
- Active:** ☒

At the bottom are 'Save' and 'Cancel' buttons. The footer indicates 'Zabbix 2.0.6 Copyright 2001-2013 by Zabbix SIA' and 'Connected as 'Admin' from 'ZBX Main''.

A form-based authentication, on the other hand, relies on the ability of the client (a Zabbix server, in this instance) to keep the session cookies, and is triggered when the said client submits a form with the authentication data. While defining a scenario, you'll need to dedicate a step just for the authentication. To know which form fields you'll need to submit, look at the HTML source of the page containing the authentication form. In the following example, we'll look at the Zabbix authentication page. Every form will be slightly different, but the general structure will be largely the same (here, only the login form in an abbreviated manner is shown).

```
<form action="index.php" method="post">
<input type="hidden" name="request" class="input hidden" value="" />
  <!-- Login Form -->
    <div >Username</div>
    <input type="text" id="name" name="name" />
```

```
<div >Password</div>
<input type="password" id="password" name="password" />
<input type="checkbox" id="autologin" name="autologin"
  value="1" checked="checked" />
<input type="submit" class="input" name="enter"
  id="enter" value="Sign in" />
</form>
```

You need to take note of the input tags and their name options, because these are the form fields you are going to send to the server to authenticate. In this case, the username field is called `name`, the password field is called `password`, and finally, the submit field is called `enter` and has the value `Sign in`.

We are now ready to create a scenario, as shown in the following screenshot:

The screenshot shows a web monitoring configuration window titled "CONFIGURATION OF WEB MONITORING". It has two tabs: "Scenario" and "Steps", with "Steps" currently selected. The form contains the following fields and controls:

- Application:** A text input field containing "Web Scenarios" and a "Select" button to its right.
- Name:** A text input field containing "Zabbix frontend".
- Authentication:** A dropdown menu currently set to "None".
- Update interval (in sec):** A text input field containing "60".
- Agent:** A dropdown menu currently set to "Mozilla Firefox 8.0".
- Variables:** A large, empty text area for defining variables.
- Active:** A checkbox that is currently checked.

At the bottom of the window, there are two buttons: "Save" and "Cancel".

We will then define the authentication step, as shown in the following screenshot:

The screenshot shows a dialog box titled "Step of scenario". It contains several input fields for defining a step:

- Name:** Login
- URL:** http://localhost/zabbix/index.php
- Post:** name=Admin&password=zabbix&enter="Sign In"
- Timeout:** 15
- Required string:** Monitoring
- Required status codes:** 200

At the bottom of the dialog box, there are two buttons: "Add" and "Cancel".

From now on, every URL that you'll check, or a form that you'll submit, will be in the context of an authenticated session, assuming the login process was successful, of course.

Logging out

One common mistake when it comes to web monitoring is that the authentication part is taken care of at the start of a scenario, but rarely at the end during logout. If you don't log out of a website, depending on the system used to keep track of the logged-in users and active sessions, a number of problems may arise.

Active sessions usually range from a few minutes to a few days. If you are monitoring the number of logged-in users, and your session's timeouts are on the longer side of the spectrum, every login scenario would add to the number of active users reported by the monitoring items. If you don't immediately log out at the end of the scenario, you may, at the very least, end up with monitoring measurements that are not really reliable, and they would show a lot of active sessions that are really just monitoring checks.

Furthermore, every new session uses up a small amount of computing resources, whether it's disk space or memory. If you create a large number of sessions in a short time, due to frequent checks, you could end up significantly degrading the website's performance.

At the very worst, your identity manager and authentication backend may not be equipped to handle a great number of non-expiring sessions and suddenly stop working, bringing your whole infrastructure to a grinding halt. We can assure you that these are not hypothetical situations, but real-life episodes that occurred in the authors' own experience.

At any rate, you certainly can't go wrong by adding a log-out step to every web scenario that involves a log in. You'll make sure that your monitoring actions won't cause any unforeseen problem, and at the very least, you will also test the correct functioning of your session's tear-down procedures. Log-out steps are also usually quite easy, as they normally involve just a GET request to the correct URL. In the case of the Zabbix frontend, you would create the following final step (as shown in the following screenshot) before ending the scenario:

The screenshot shows a dialog box titled "Step of scenario". It contains several input fields for configuring a new step:

- Name:** A text box containing the word "Logout".
- URL:** A text box containing the URL "http://localhost/zabbix/index.php?reconnect=1".
- Post:** A large, empty text area for specifying POST data.
- Timeout:** A text box containing the number "15".
- Required string:** An empty text box.
- Required status codes:** A text box containing the number "200".

At the bottom of the dialog, there are two buttons: "Add" (highlighted with an orange border) and "Cancel".

Aggregated and calculated items

Until now, every item type described in this chapter can be considered as a way of getting raw measurements as single data points. In fact, the focus of the chapter has been more on setting up Zabbix to retrieve different kinds of data, than on what is actually collected. This is because on one hand, a correct setup is crucial for effective data gathering and monitoring, while on the other hand, the usefulness of a given metric varies wildly across environments and installations, depending on the specific needs that you may have.

When it comes to aggregated and calculated items though, things start to become really interesting. Both types don't rely on probes and measurements at all; instead, they build on existing item values to provide a whole new level of insight and elaboration on the data collected in your environment.

This is one of the points where Zabbix's philosophy of decoupling measurements and triggering logic really pays off, because it would be quite cumbersome otherwise to come up with similar features, and it would certainly involve a significant amount of overhead.

The two types have the following features in common:

- Both of them don't make any kind of checks (agent-based, external, SNMP, JMX, or otherwise) but directly query the Zabbix database to process the existing information.
- While they have to be tied to a specific host because of how Zabbix data is organized, this is a loose connection compared to a regular item. In fact, you could assign an aggregated item to any host regardless of the item's specifics, although it's usually clearer and easier if you define one or more simple, dedicated hosts that will contain aggregated and calculated items, so that they'll be easier to find and reference to.
- Aggregated and calculated items only work with numeric data types—there's no point in asking for the sum or the average of a bunch of text snippets.

Aggregated items

The simplest of the two types discussed here, aggregated items, can perform different kinds of calculations on a specific item that is defined for every host in a group. For every host in a given group, an aggregated item will get the specified item's data (based on a specified function), and then apply the group function on all of the values collected. The result will be the value of the aggregated item measurement at the time that it was calculated.

To build an aggregated item, you first need to identify the host group you are interested in, and then identify the item, shared by all the group's hosts, which will form the basis of your calculations. For example, let's say that you are focusing on your web application servers and you want to know something about the active sessions of your Tomcat installations. In this case, the group would be something like `Tomcat Servers`, and the item key would be `jmx["Catalina:type=Manager,path=/,host=localhost",activeSessions]`.

Next, you need to decide how you want to retrieve every host's item data. This is because you are not limited to just the last value, but can perform different kinds of preliminary calculations. Except for the `last` function, which indeed just retrieves the last value from the item's history, all the other functions take a period of time as a further argument.

Function	Description
<code>avg</code>	The average value in the specified time period
<code>sum</code>	The sum of all values in the specified time period
<code>min</code>	The minimum value recorded in the specified time period
<code>max</code>	The maximum value recorded in the specified time period
<code>last</code>	The latest value recorded
<code>count</code>	The number of values recorded in the specified time period

What you now have is a bunch of values that still need to be brought together. The following table explains the job of the group function:

Function	Description
<code>grpavg</code>	The average of all the values collected
<code>grpsum</code>	The sum of all values collected
<code>grpmin</code>	The minimum value in the collection
<code>grpmax</code>	The maximum value in the collection

Now that you know all the components of an aggregated item, you can build the key; the appropriate syntax is as follows:

```
groupfunc["Host group","Item key",itemfunc,timeperiod]
```



The Host group part can be defined locally to the aggregated item definition. If you want to bring together data from different hosts that are not part of the same group, and you don't want to create a host group just for this, you can substitute the host group name with a list of the hosts—["HostA, HostB, HostC"].

Continuing with our example, let's say that you are interested in collecting the average number of active sessions on your Tomcat application server's every hour. In this case, the item key would look as follows:

```
grpavg["Tomcat servers",  
  "jmx["Catalina:type=Manager,path=/,  
  host=localhost",activeSessions]", avg, 3600]
```



You could also use 1h or 60m as a time period if you don't want to stick to the default of using seconds.

Using the same group and a similar item, you would also want to know the peak number of concurrent sessions across all servers, this time every five minutes, which can be done as follows:

```
grpsum["Tomcat servers",  
  "jmx["Catalina:type=Manager,path=/,host=localhost",maxActive]",  
  last, 0]
```

Simple as they are, aggregated items already provide some useful functionality, which would be harder to match without a collection of measurements as simple data that is easily accessible through a database.

Calculated items

This item type builds on the concept of item functions expressed in the previous paragraphs, and takes it to a new level. Unlike aggregated items, with calculated ones you are not restricted to a specific host group, and more importantly, you are not restricted to a single item key. With calculated items, you can apply any of the functions available for the trigger definitions to any item in your database, and combine different item calculations using arithmetic operations. As with other item types that deal with specialized pieces of data, the item key of a calculated item is not used to actually define the data source, but still needs to be unique so that you can refer to the item in triggers, graphs, and actions. The actual item definition is contained in the `formula` field, and as you can imagine, it can be as simple or as complex as you need.

Keeping with our Tomcat server's example, you could have a calculated item that gives you a total application throughput for a given server as follows:

```
last(jmx["Catalina:type=GlobalRequestProcessor,name=http-
8080",bytesReceived]) +
last(jmx["Catalina:type=GlobalRequestProcessor,name=http-
8080",bytesSent]) +
last(jmx["Catalina:type=GlobalRequestProcessor,name=http-
8443",bytesReceived]) +
last(jmx["Catalina:type=GlobalRequestProcessor,name=http-8443",
bytesSent]) +
last(jmx["Catalina:type=GlobalRequestProcessor,name=jk-8009",
bytesReceived]) +
last(jmx["Catalina:type=GlobalRequestProcessor,name=jk-
8009",bytesSent])
```

Or you could be interested in the ratio between the active sessions and the maximum number of allowed sessions, so that later you could define a trigger based on a percentage value instead of an absolute one as follows:

```
100*last(jmx["Catalina:type=Manager,path=/,
host=localhost",activeSessions]) /
last(jmx["Catalina:type=Manager,path=/,host=localhost",
maxActiveSessions])
```

As previously stated, you don't need to stick to a single host either, in your calculations.

The following is how you could estimate the average number of queries on the database per single session, on an application server, every three minutes:

```
avg(DBServer:mysql.status[Questions], 180) /  
  avg(Tomcatserver:Catalina:type=Manager,path=/,host=localhost",  
    activeSessions], 180)
```

The only limitation with calculated items is that there are no easy group functions like those available to aggregated items. So while calculated items are essentially a more powerful and flexible version of aggregated items, you still can't dispense with the latter, as you'll need them for all group-related functions.

Despite this limitation, as you can easily imagine, the sky is the limit when it comes to calculated items. Together with aggregated items, these are ideal tools to monitor the host's group performances such as clusters or grids, or to correlate different metrics on different hosts that contribute to the performance of a single service.

Whether you use them for performance analysis and capacity planning, or as the basis of complex and intelligent triggers, or both, the judicious use of aggregated and calculated items will help you to make the most out of your Zabbix installation.

Summary

In this chapter, we delved into various aspects of item definitions in Zabbix. At this point, you should know the main difference between a Zabbix item and a monitoring object of other products, and why Zabbix's approach of collecting simple, raw data instead of monitoring events, is a very powerful one. You should also know the ins and outs of monitoring the data flow traffic, and how to affect it based on your needs and environment. You should be comfortable moving beyond the standard Zabbix agent when it comes to gathering data, and be able to configure your server to collect data from different sources—database queries, SNMP agents, web pages, JMX consoles, and so on.

Finally, you should've grasped the vast possibilities implied by the two powerful item types—aggregated and calculated.

In the next chapter, you'll learn how to present and visualize all the wealth of data you are collecting using graphs, maps, and screens.

5

Visualizing Data

Zabbix is a flexible monitoring system. Once implemented on an installation, it is ready to support a heavy workload and will help you acquire a huge amount of every kind of data. The next step is to graph your data, interpolate, and correlate the metrics between them. The strong point is that you can relate different types of metrics on the same axis of time, analyzing patterns of heavy and light utilization, identifying services and equipment that fail most frequently in your infrastructure, and capturing relationships between metrics of the connected services.

Zabbix, beyond the standard graphs facility, offers you a way to create your custom graphs, and to add them on your own template, thus creating an easy method to propagate your graphs across all the servers. Those custom graphs (and also the standard and simple) can be collected into screens. Inside Zabbix, a screen can contain different kind of information — simple graphs, custom graphs, other screens, plain text information, trigger overview, and so on.

In this chapter, you will learn how to:

- Generate custom graphs
- Create and use maps with shortcuts and nested maps
- Create a dynamic screen
- Create and set up slides for a large monitor display
- Generate an SLA reporting

As a practical example, you can think of a big datacenter, where there are different layers or levels of supports; usually the first level of support needs to have a general overview of what is happening on your datacenter, the second level can be the first level of support divided for typology of service, for example, DBA, Application Servers, and so on. Now your DBA second level of support will need the entire database related metrics, an application server specialist most probably will need all the Java metrics, plus some other standard metrics such as CPU Memory usage. Zabbix's responses to this requirement are maps, screens, and slides.

Once you create all your graphs, and have retrieved all your needed metrics and messages, you can easily create screens that collect, for instance, all the DBA related graphs plus some other standard metrics; it will be easy to create a rotation of those screens. The screen will be collected on slides and each level of support will see its groups of screens in a slide show, having an immediate qualitative and quantitative vision of what is going on.

The datacenter support is most probably the most complex slide show to implement, but in this chapter you will see how easy it is to create them. Once you have all the pieces (simple graphs, custom graphs, triggers, and so on), you can use them, and also re-use them on different visualization types. On most of the slides, for instance, all the vital parameters such as CPU, memory, swap usage, and network I/O needs to be graphed. Once done, your custom graphs can be re-used in a wide number of dynamic elements. Zabbix provides another great functionality, that is, the ability to create dynamics **maps**. A map is a graphical representation of a network infrastructure. All those features will be discussed in this chapter.

When you are finally ready to implement your own custom visualization screen, it is fundamental to bear in mind which one will be the audience, their skills or background, and their needs. Basically, be aware of what message you will deliver with your graphs.

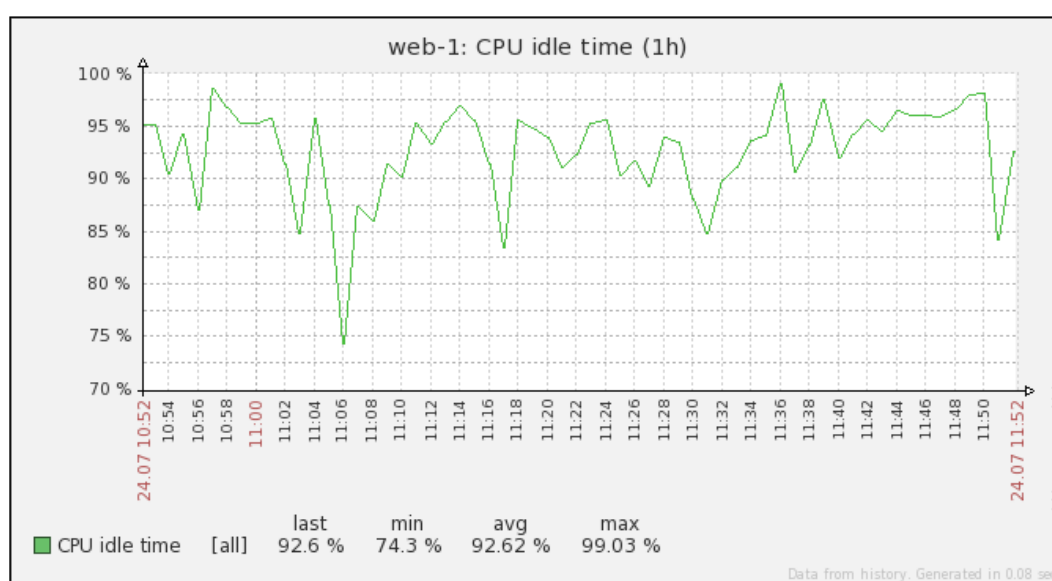
Graphs are powerful tools to transmit your message; they are a flexible instrument that can be used to give more strength to your speech as well as give a qualitative overview of your service or infrastructure. This chapter is pleasant, and will enable you to communicate using all the Zabbix graphical elements.

Graphs

Inside Zabbix, you can divide the graphs in two categories – the simple graphs and the custom graphs. Both of these are analyzed in the next section.

Analyzing simple graphs

Simple graphs in Zabbix are something really immediate since you don't need to put in a lot of effort to configure this feature. You only need to go to **Monitoring | Latest data**, eventually filter by the item name, and click on the graph. Zabbix will show you the historical graph, as shown in the following screenshot:

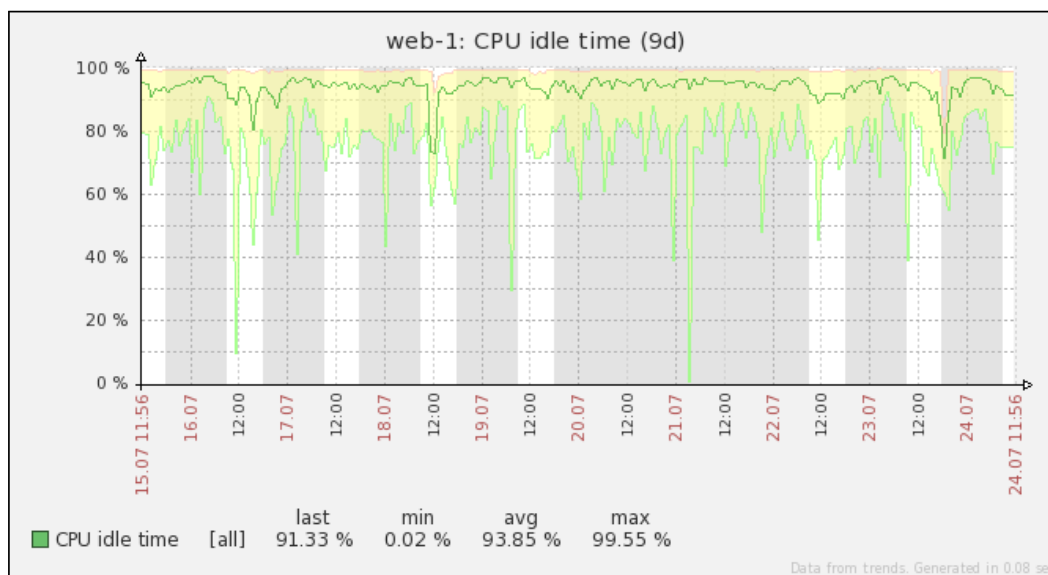


Clearly you can graph only numeric items, and all the other kinds of data such as text can't be shown on a graph. On the Latest data item, you will see instead the Graph link – a link that will show the history.



No configuration is needed, but you can't customize this graph.

On top of the graphs there is the time period selector. If you enlarge this period, you will see the aggregated data. Until the period is little and you would like to see very recent data, you will see a single line. If the period is going to enquire the database for old data, you will see three lines. This fact is tied to history and trends; since the values are contained in the history table, the graph will show only one line. Once you're going to retrieve data from the trends, the lines will be three, as shown in the following screenshot:



In the previous screenshot, we can see three lines that define a yellow area. This area is designed by the minimum and maximum values, and the green line represents the mean value. A quite complete discussion about trends/history tables has been discussed in *Chapter 1, Deploying Zabbix*. Here it is important to have all those three values graphed.

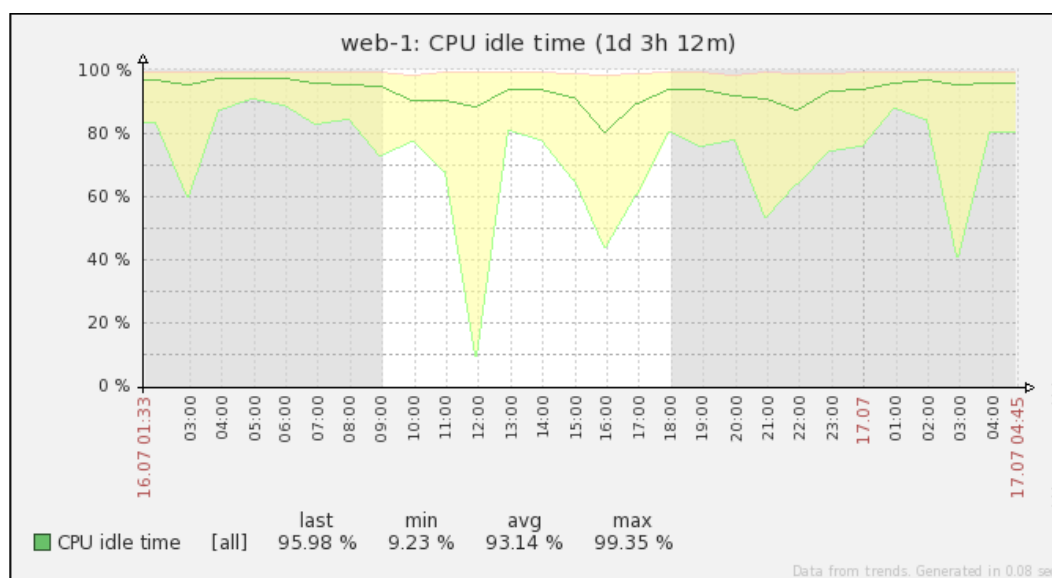


Longevity of an item into the history is defined into the item itself in the field `Keep history (in days)` as well as the persistence on the trends table is defined in the field `Keep trends (in days)`.

In the following screenshot, you can see how the mean values may vary with respect to the minimum and maximum value. In particular, it is interesting to see how the mean value remains almost the same at **12:00** also. You can see quite an important drop in the **CPU idle time** (light green line), that didn't influence the mean value (green line) too much since most likely it was only a small and quick drop, so it is basically lost on the mean value, but not on our graph since Zabbix preserves the minimum and maximum value.



Graphs show the working hours with a white background, and the non-working hours in gray (using the original template); working time is not displayed if the graph needs to show more than three months.



Simple graphs are intended just to graph some on spot metrics, and check a particular item. Of course, it is important to interpolate the data, for instance on the CPU you have different metrics and it is important to have all of them.

Analyzing custom graphs

We have only discussed the graph components here rather than the full interaction functionality and importance of them to see historical trends, or delving into some specific time period on some particular date. Zabbix offers the custom graphs functionality – these graphs need to be created and customized by hand. For instance, there are some predefined graphs on the standard **Template OS Linux**. To create a custom graph, you need to go to **Configuration | Hosts (or Templates)**, and click on **Graphs** and then on **Create graph**.



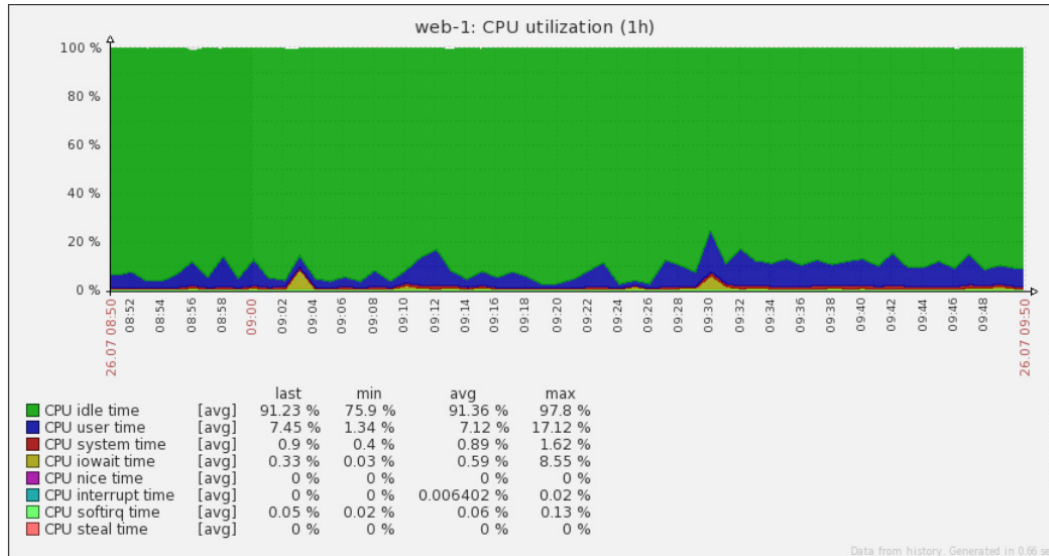
General graphs should be created into templates so that they can be easily applied to a group of servers. An example is the graph of CPU utilization on Template OS Linux. This one is quite general; it is composed of several metrics aggregated and is nice to have across all your Linux servers.

Graphs on Zabbix are really a strong point of the monitoring infrastructure. Inside this custom graph, you can choose if you want to show the working time and the legend, by using different kind of graphs. The details of the CPU Utilization graph is shown in the following screenshot:

Name	Function	Y axis side	Colour	Action
1: Template OS Linux: CPU steal time	avg	Left	FF5555	Remove
2: Template OS Linux: CPU softirq time	avg	Left	55FF55	Remove
3: Template OS Linux: CPU interrupt time	avg	Left	009999	Remove
4: Template OS Linux: CPU nice time	avg	Left	990099	Remove
5: Template OS Linux: CPU iowait time	avg	Left	999900	Remove
6: Template OS Linux: CPU system time	avg	Left	990000	Remove
7: Template OS Linux: CPU user time	avg	Left	000099	Remove
8: Template OS Linux: CPU idle time	avg	Left	009900	Remove

[Add](#)

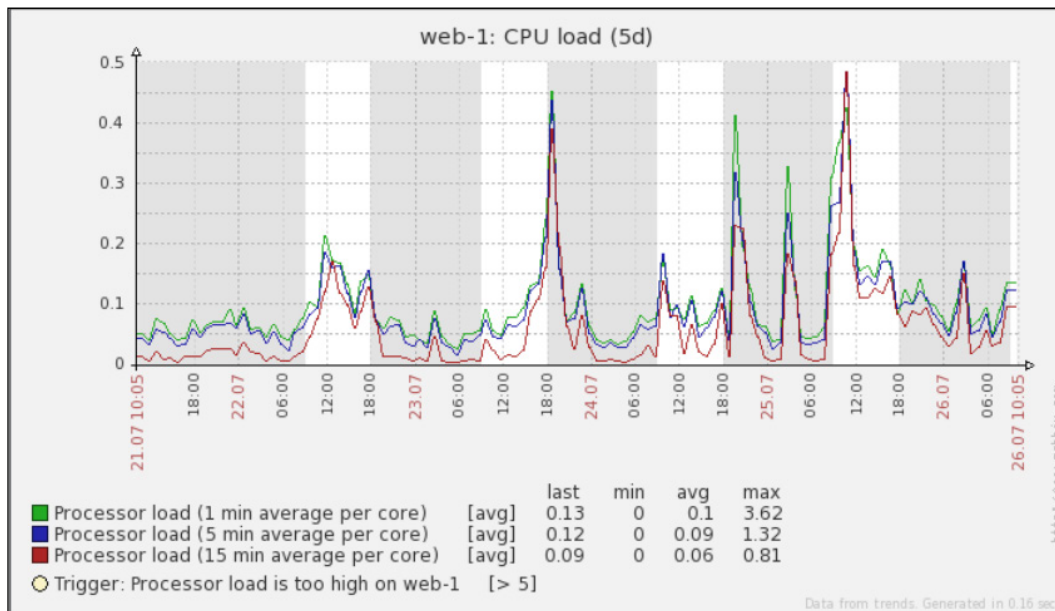
As you can see the following graph is stacked, and shows the legend of the x axis defined with a fixed y axis scale. In this particular case, it doesn't make any sense to use a variable for the minimum or maximum values of y axis since the sum of all the components represent the whole CPU, and each component is a percentage. Since a stacked graph represents the sum of all the stacked components, this one will always be 100 percent as shown in the following screenshot:



There are some considerations when it comes to triggers and working hours. These are only two checks, but they change the flavor of the graph. In the previous graph, the working hours are displayed on the graph but not triggers, mostly because there aren't triggers defined for those metrics. The working hours, as mentioned earlier, are displayed in white. Displaying working hours is really useful in all the cases where your server has two different life cycles or serves two different tasks. As a practical example, you can think about a server placed in New York that monitors and acquires all the market transactions of the U.S. Market. If the working hours — as on this case — will be coincident with the market opening hours, the server will most probably acquire data for most of the time. Think about what will happen if the same trading company works in the Asian market; most probably they will enquire the server in New York to see what happened while the market was open. Now in this example, the server will provide a service for two different scenarios, and have the working hours displayed in a graph, which can be really useful.

💡 Displaying the working time on graphs is useful.
See if your trigger goes on fire in this period.

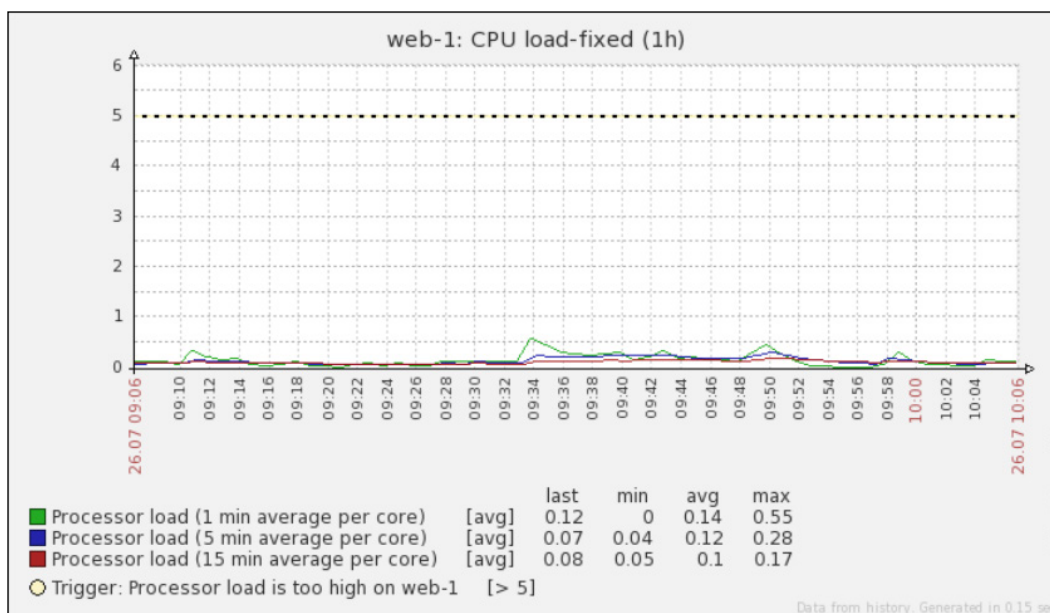
Now if you want to display the triggers in your graph, you only need to mark the **Show triggers** checkbox and all the triggers defined will be displayed on the graph. Now it can happen that you don't see any lines about the triggers in your graph, for instance, looking at the following screenshot:



Now where is your expected trigger line? Well, it is simple. Since the trigger is defined for a processor load greater than five, to display this line you need to make a few changes in this graph, in particular the **Y axis MIN value** and **Y axis MAX value** fields. In the default predefined CPU load graph, the minimum value is defined as zero and the maximum value is calculated. They both need to be changed as follows:

Y axis MIN value	Fixed	▼	0.0000
Y axis MAX value	Fixed	▼	6.0000

Now refresh your graph. Finally you will see the trigger line, which wasn't visible in the previous chart because the CPU was almost idle, and the trigger threshold was too high and not displayed due to the auto scaling on y axis.

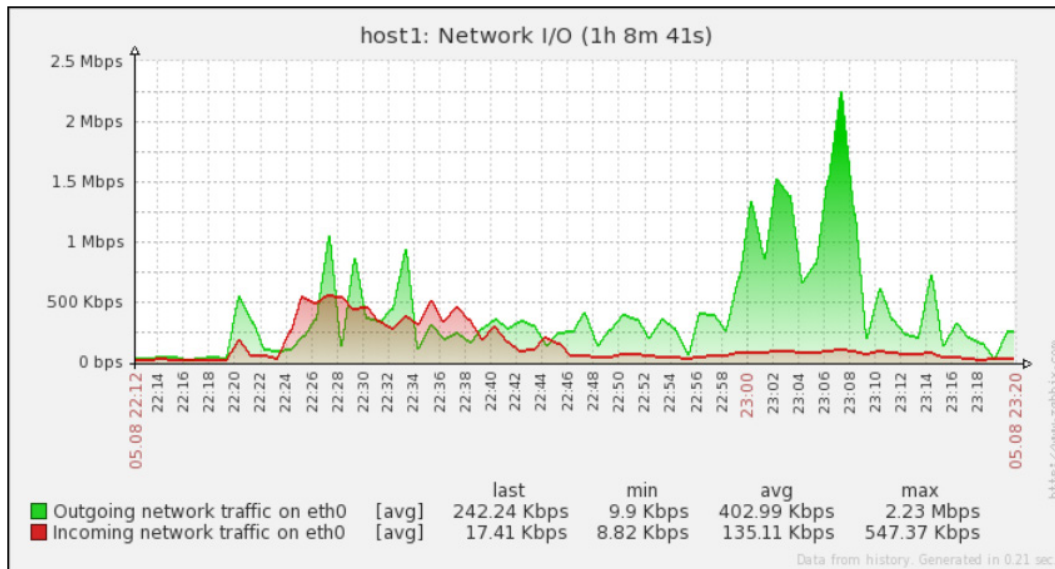


As you probably already noticed, Zabbix doesn't display periods shorter than an hour. The minimum graph's period is about one hour.

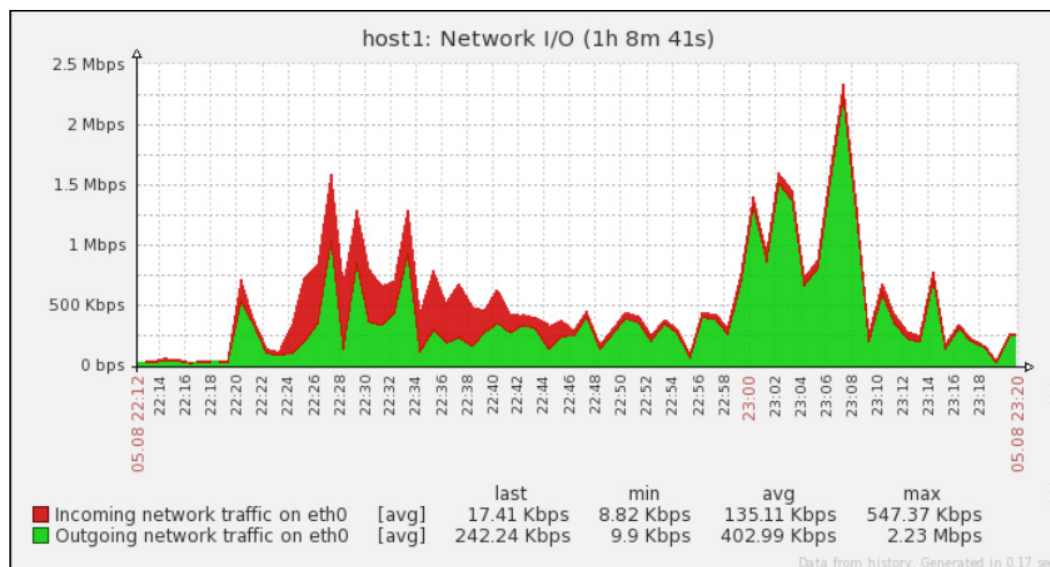
Zabbix supports the following kinds of custom graphs:

- Normal
- Stacked
- Pie
- Exploded

Zabbix also supports different kinds of drawing styles. Graphs that display the network I/O, for instance, can be made using gradient lines; this will draw an area with a marked line for the border, so you can see the incoming and outgoing network traffic on the same scale. An example of this kind is shown in the following screenshot which is easy to read. Since you don't have the total throughput to have graphed the total amount from the incoming packet, the outgoing packet is the better one to be chosen for a stacked graph. In stacked graphs, the two areas are summarized and stacked, so the graph will display the total consumed bandwidth.



To highlight the difference between a normal graph and a stacked one, the following screenshot displays the same graph during the same time period, so it will be easier to compare them:



As you can see, the peaks and the top line are made aggregating the network input and output of your network card. The preceding graph represents the whole network traffic handled by your network card.

Reviewing all the combinations of graph properties

Zabbix is a quite flexible system and the graphs are really customizable, to better explore all the possible combination of attributes, and parameters that can be customized. All the possible combinations of graphs attributes are reviewed in the following table:

Parameter	Description
Name	Graph name (Note: needs to be unique)
Width	Graph width in pixels
Height	Graph height in pixels
Graph type	Normal (values displayed as lines, filled region, bold lines, dots, dashed lines, and gradient lines) <ul style="list-style-type: none"> Stacked values are displayed as stacked areas Pie values are displayed as a pie Exploded values displayed as pie but exploded
Show legend	If checked, the graph will display the legend.

Parameter	Description
Show working time	If checked, the non-working hours will be in gray and working hours in white background.
Show triggers	If checked, a single trigger line is displayed (Note: this is not available for pie and exploded).
Percentile line (left/right)	Note: only available on normal graphs. If checked, it displays a line where the value falls under the percentage (for example, for 90 it will display a line where 90 percent of the values fall under).
Y axis MIN/MAX value	Set the minimum and maximum value for the y axis, which can be any of the following: <ul style="list-style-type: none">• Calculated: The minimum and maximum values will be calculated on the basis of the displayed area• Fixed: The fixed value will be used as per maximum or minimum• Item: The last value of the selected item will be used as minimum/maximum
3D View	It displays the graphs in 3D (Note: only available for pie and exploded pie graphs).
Sort order	This represents the priority for a particular item over the other and is useful to give priority to the region displayed, for example, in front of or behind the other items. Here you can drag-and-drop items in the right order. Note: zero is the first processed item; Zabbix supports up to 100 items.
Name	Name of the item is displayed here. The metrics name is composed in the form of <source> : <metric_name>. This means that if you are inside the host configuration, you will see <hostname>:<metric_name>. If you are creating the graphs inside a template, you will see <template_name>:<metric_name>.
Type	Note: this is available only on pie and exploded pie graphs. <ul style="list-style-type: none">• Simple• Graph sum

Parameter	Description
Function	What value to display if more than one are present; it can be one of the following: <ul style="list-style-type: none"> • All: Minimum, average, and maximum values • Min: Only minimum values • Avg: Only average values • Max: Only maximum values
Draw style	Available only for normal graph <ul style="list-style-type: none"> • Line • Filled region • Bold line • Dot • Dashed line
Y axis side	Note: available only on stacked and normal graphs, and defines the y axis side for each elements.
Colour	Other than the standard displayed on the palette, you can set all the colors that you want in the RGB hex format.



You can easily play with all those functionality and attributes. In Version 2.0 of Zabbix, you have a **Preview** tab that is really useful when you're configuring a graph inside a host. If you're defining your graph at a template level, this tab is useless because it doesn't display the data. When you are working with templates, it is better to use two windows to see in real time by refreshing (the *F5* key) the changes directly against the host that inherits the graphs from the template.

All the options previously described are really useful to customize your graphs, as you have understood that graphs are really customizable and flexible elements.



You can display only three trigger lines, and if the graph is less than 120 pixels, triggers are not displayed; so take care to properly set up your graphs and check all the changes.

Visualizing the data through maps

Zabbix provides a powerful element to visualize data and a topological view in Zabbix, which will help you to create maps. Maps are a graphical representation of your physical infrastructure, where you can display your server, network components, and the interconnection between them.

The great thing is that maps on Zabbix are completely dynamic, which means that you will see your active warning, issues, and triggers represented on the map, with different icons, or color and labels. This is a powerful representation of the status of art of your datacentre, or of the service that you're representing. The elements that you can put in a map are as follows:

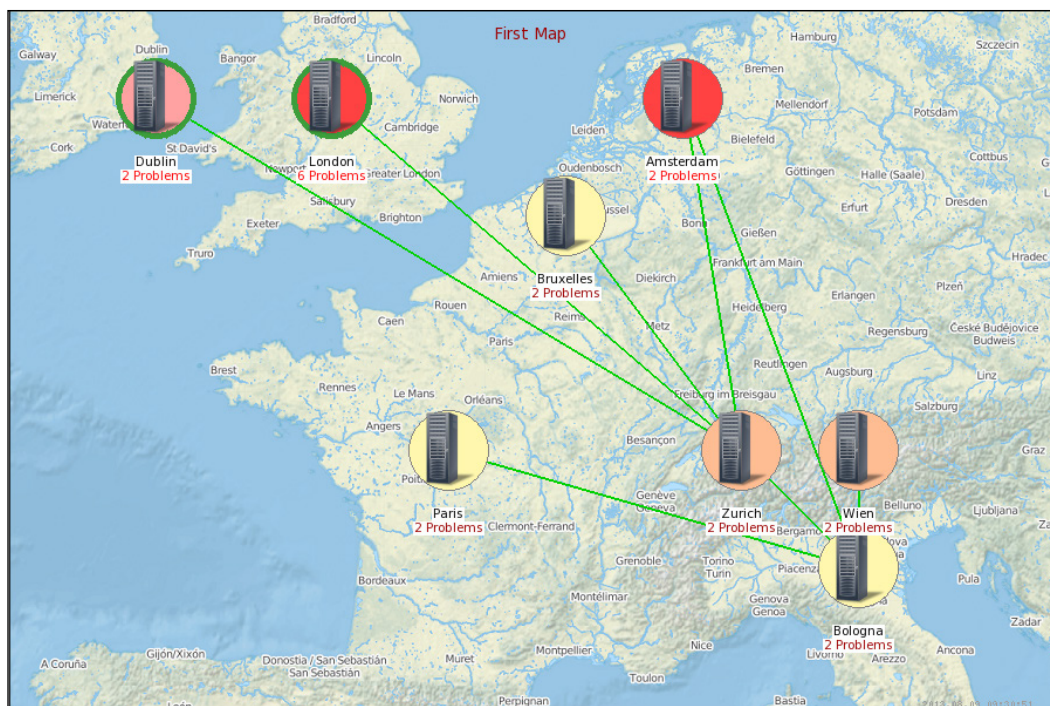
- Host
- Host groups
- Triggers
- Images
- Maps

All those elements are dynamically updated by triggers or using macros, hence providing a complete status of the maps and their elements.

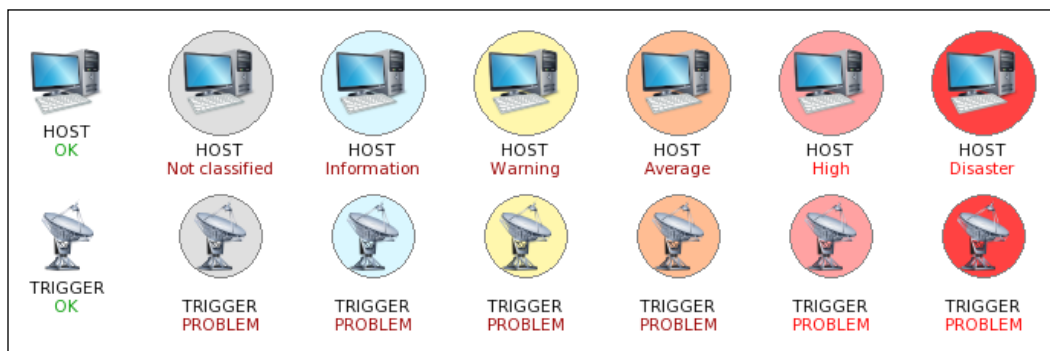


To enable a user to create, configure, or customize maps, the user needs to be of the *Zabbix admin* type. This means that there isn't a role dedicated to map creation. Also, the user needs to have a read/write permission on all the host that he needs to put into a map. This means that there isn't a way to restrict the privileges to the map creation only, but you can limit the administration to a certain number of hosts included in a group.

An example of a map that you can easily produce with Zabbix is shown in the following screenshot:



In the preceding screenshot, you can see that there is a lot of graphical combination of icons, round backgrounds, and information. To better understand what this map represents, it is important to see how Zabbix treats hosts and triggers in a map. In the following screenshot, you can see all the possible combinations of trigger severity and status change:



The preceding screenshot illustrates, from left to right, the following:

- A host that doesn't have a trigger on fire
- A host with a trigger severity **Not classified** in the alarm
- A host with a trigger severity **Information** in the alarm
- A host with a trigger severity **Warning** in the alarm
- A host with a trigger severity **Average** in the alarm
- A host with a trigger severity **High** in the alarm
- A host with a trigger severity **Disaster** in the alarm

The trigger line follows the same classification.

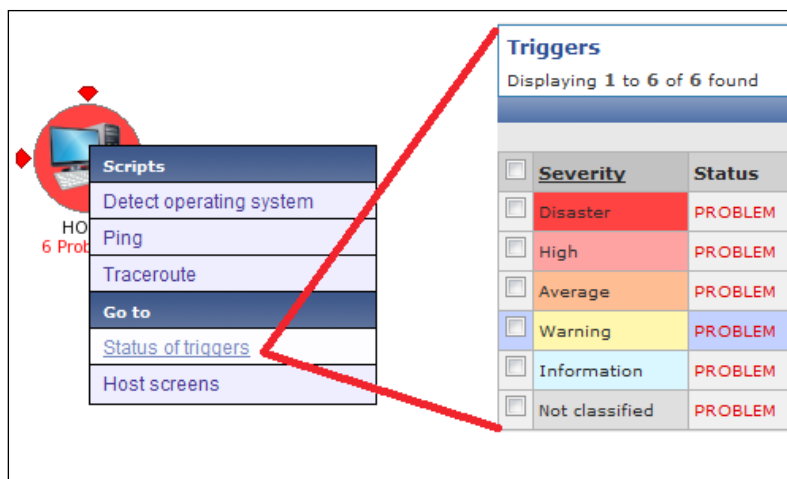


The trigger severity is expressed with the background color and not with the name that you see under the **HOST** label. The label in red color is the name of the trigger. In the preceding screenshot, triggers are callers, right like their classification, to simplify make the picture more verbose. Please notice that in case of TRIGGER representation, right after the **TRIGGER** label is displayed, the trigger status is displayed, and not the trigger name that is on fire, as in the case of HOST.

Now if a trigger changed recently, its status will be displayed as shown in the following screenshot:



Please note that in this case the icon is shown with the arrows because it just changed the status. The following screenshot shows that there are six problems, and drilling down which ones those are:



As you can see, there are different triggers with problems. The one that has the most critical severity is the one that gives the color to the circle around the icon. Once all the triggers are acknowledged, the icon will show a green circle around it, as shown in the following screenshot:



The second icon displays a detail of all the problem that your host is facing, and how many of them are not acknowledged, so you have an immediate status of how many issues are under control, and how many are new.

The third icon with a square background is a host that has been disabled, represented in gray; it will be in red once unreachable.

Creating your first Zabbix Map

Map configuration can be easily reached by navigating to **Configuration | Maps | Create map**. The resulting window that you will see is shown in the following screenshot:

Map

Name

Width

Height

Background image

Automatic icon mapping [show icon mappings](#)

Icon highlight ☐

Mark elements on trigger status change ☐

Expand single problem ☐

Advanced labels ☒

Host group label type

Host label type

Trigger label type

Map label type

Image label type

Icon label location

Problem display

URLs

Name	URL	Element
<input type="text"/>	<input type="text"/>	<input type="text" value="Host"/>

[Add](#) [Remove](#)

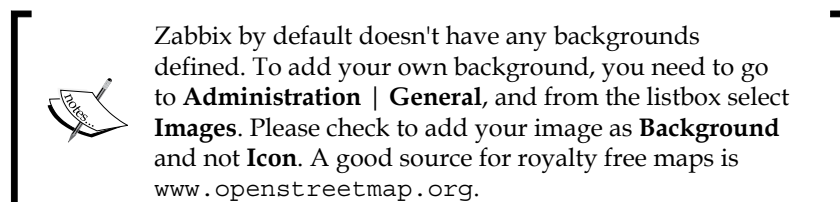
Most of the properties are quite intuitive; the **Name** field needs to be a unique name, **Width** and **Height** are expressed in pixels.



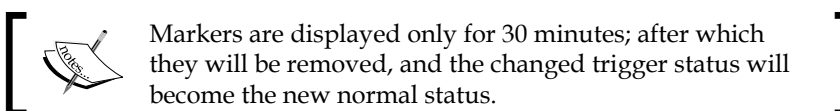
If you define a large size, and in the second time you want to reduce it, it is possible that some of your hosts will fall outside the map, and are no longer visible. Don't be scared; nothing is lost. They are still inside the map, only not displayed. You can restore it to the original size, and they will appear again.

Now we will take a look at all the other following parameters:

- **Background image:** In the background image field you can define your map's background, choosing between the loaded backgrounds.



- **Automatic icon mapping:** This flag enables a user-defined icon mapping for a certain host inventory field. This can be defined by navigating to **Administration | General | Icon mapping**.
- **Icon highlight:** This is the flag responsible to generate the round background around the icon, with the same color of the most critical severity trigger.
- **Mark elements on trigger status change:** This flag is responsible to highlight a trigger status change (with the red triangle shown earlier in the screenshot displaying the status change).



- **Advanced labels:** This check enables you to customize the label's type for all the elements that you can put in a map. So for each one of those items – host, host group, trigger, map, and image – you can customize the label type.

The possible label types are as follows:

- **Label:** Icon label
- **IP Address:** IP address (note: available only on host)
- **Element name:** Element name, such as hostname
- **Status only:** This is only for the status, so OK/PROBLEM
- **Nothing:** No label at all
- **Custom label:** A free text area (note: macros are allowed)
- **Icon label location:** This field defines where by default you will see all the labels. This can be selected among the following values: **Bottom**, **Left**, **Right**, and **Top**.

- **Problem display:** This listbox permits you to choose between the following:
 - **All:** A complete problem count will be displayed
 - **Separated:** This displays the unacknowledged problem count separated as a number of the total problem count
 - **Unacknowledged only:** With this selected, only the unacknowledged problem count will be displayed
- **URLs:** Here, a URL for each kind of element can be used with a label. This label is a link, and here you can use macros, for example, {MAP.ID}, {HOSTGROUP.ID}, {HOST.ID}, {TRIGGER.ID}.

Some important considerations about macros and URLs

The URL section is powerful, but here an example is needed, because the usage is not so intuitive and simple.

Now if you see a trigger on fire or an alarm that is escalating, most probably the next action that you will do is to check the latest data of your host or jump to a screen that will group the graphs, triggers, and data that you need to check to have an idea of what is happening, and do a first-level analysis. In the practical case of a first level support, once a server is highlighted and shows triggers with problems, it can be useful to have a link that will go straight ahead to the latest data of that host and also the screen. To automate this and reduce the number of clicks, you can simply copy the link of the desired page, for instance the link to the latest data would be something as follows:

```
http://<YOUR-ZABBIX-SERVER>/zabbix/latest.  
php?sid=eec82e6bdf51145f&form_refresh=1&groupid=2&hostid=10095
```

Now looking into the URL to automate the jump to the latest data, you need to replace the variant part of the URL with the macros wherever available.

The `sid` value in the URL represents the session ID; it is passed to avoid the one-click attack also known as **session riding**. This field can be removed. The `groupid` value in the specific latest data example can be omitted, so the URL can be reduced as follows:

```
http://<YOUR-ZABBIX-SERVER>/zabbix/latest.php?form_  
refresh=1&hostid=10095
```

Now the link is easy to generate. You can simply replace the `hostid` value with the macro `{HOST.ID}` as follows:

```
http://<YOUR-ZABBIX-SERVER>/zabbix/latest.php?form_refresh=1&hostid={HOST.ID}
```

And configure the URL as shown in the following screenshot:

URLs	Name	URL	Element	
	General Screen	refresh=1&fullscreen=0&hostid={HOST.ID}	Host	Remove
	Latest Data	/zbx/zabbix/latest.php?hostid={HOST.ID}	Host	Remove
	Add			

In the preceding screenshot, you can see that there is a link configured to **General Screen** that collects the most important graphs. The following is the URL generated from the screen page of a particular host:

```
http://<ZABBIX-SERVER>/zabbix/screens.php?sid=eec82e6bdf51145f&form_refresh=1&fullscreen=0&elementid=17&groupid=2&hostid=10094&period=3600&stime=20140807161304
```

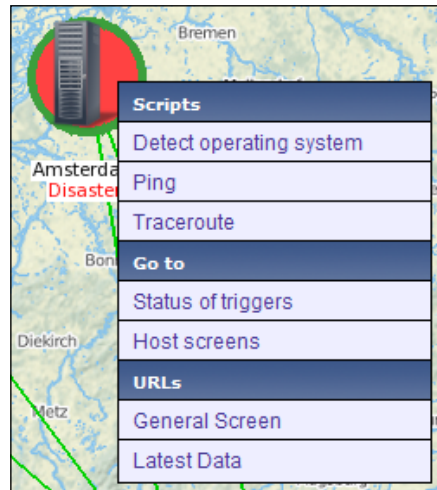
This time again, you can omit the `sid` value in the preceding URL, since it specifies a period. If this parameter is absent, you will be taken to a screen that displays the last hour of data. You can also remove the `stime`, `groupid`, and `elementid` values. The reduced URL will be as follows:

```
http://<ZABBIX-SERVER>/zabbix/screens.php?form_refresh=1&fullscreen=0&hostid=10094&groupid=2
```

Now, to make it dynamic, you need to replace the values of `hostid` and `groupid` with the macros, as shown in the following URL:


```
http://<ZABBIX-SERVER>/zabbix/screens.php?form_refresh=1&fullscreen=0&hostid={HOST.ID}&groupid={HOSTGROUP.ID}
```

The result of this customization is shown in the following screenshot:



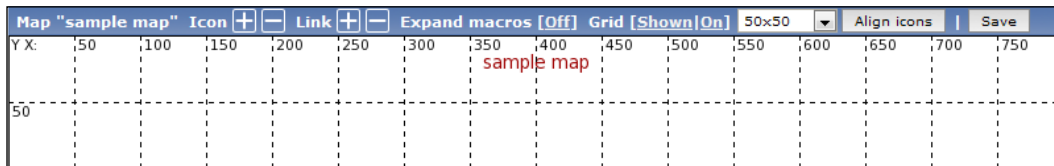
As you can see, by clicking on the host that has issues, you have two new shortcuts other than **Latest Data** and **General Screen**, with a link dynamically created for each host.

This kind of behavior allows you to create a master-detail view. In this case, the master is your map, and the detail can be for instance, the screen or the latest data window. You can create some custom menus that can run a script, or bring you directly to the trigger status or the Host screen.

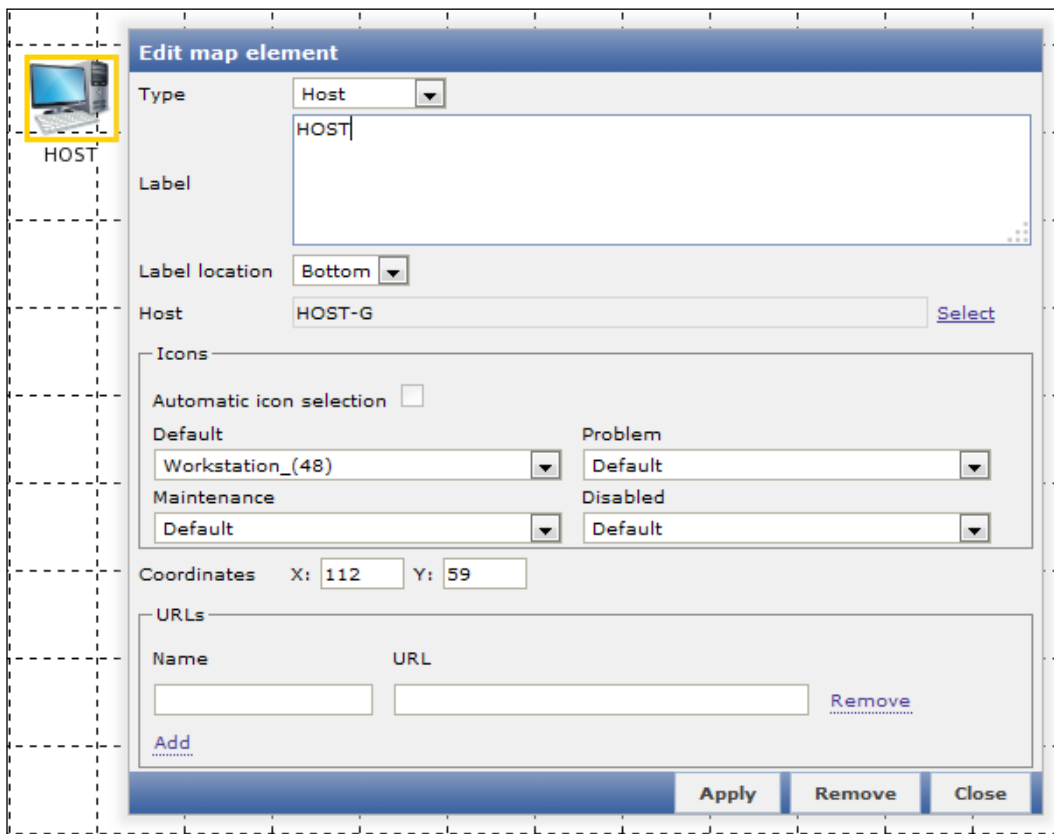
[ Here you can add some more scripts to run against the host. To add another script (and see it in the **Scripts** section), you need to go to **Administration | Scripts**. This will take you to the script's administration panel.]

Finally inside the map

Once you have completed this setup, you can begin the nice part of the configuration. Once inside the map, the menu that you will find is quite simple and user-friendly, as shown in the following screenshot:



In the map, you can add an element by clicking on the + sign, and remove it by clicking on the - sign. The element will appear in the upper-left corner of the map. By clicking on that icon, a configuration panel will appear, as shown in the following screenshot:



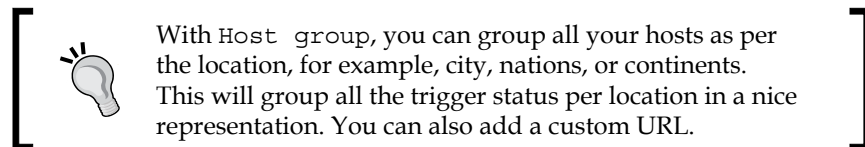
The element type by default is **Icon**. In the preceding screenshot, it is marked as **Host**, but it can be one of the following:

- **Host**: This icon will represent the status of all triggers of the selected host
- **Map**: This icon will represent the status of all the elements of the map
- **Trigger**: This icon will represent the status of the single trigger selected
- **Host group**: This icon will represent the status of all the triggers of all hosts that belong to the selected group
- **Image**: This icon will just be an image not linked to any source (trigger host and so on)

The **Label** section is another strong point of the element. Here you can freely write a normal text or, use macros.

The next field may vary depending on what you choose as the element type and can be one of the following:

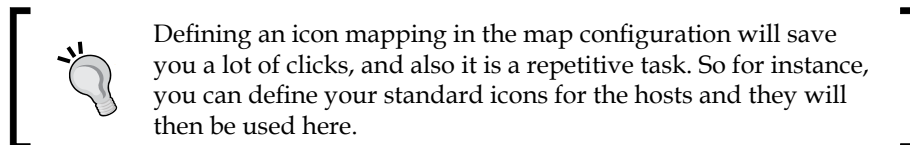
- **Host**: This selects the host
- **Map**: This selects the map
- **Trigger**: This selects the trigger
- **Host group**: This selects the host group
- **Icon** (default): This selects the icon to be used.



Host and triggers have already been covered, and are quite intuitive to understand. Probably it is not immediately understood why we should insert a map inside a map. An efficient usage of this scenario is that you can produce a nice drill down having a general map view that gathers together all the sub maps, detailed per location or nation. This helps you produce a drill down until the final destination, for instance, try to think about a drill down that comes from nations, down to the city, and deep into the datacenter, ending on the rack where your server is contained.

The **Icon** element inside a map is an image that can have a URL associated with it. Their function is to add some graphic element to your map that contains a URL, and have the shortcuts directly on your own map.

Right after that, there is the **Icons** section. Here, if you checked the **Automatic icon selection** checkbox, the icon mapping (defined in the map configuration) will be used to choose the icons to be used.



If you haven't defined an icon mapping, or if you want to use an item different from the previous choice, you can specify the icons that will be used on those cases, which can be one of the following:

- **Default**
- **Problem**
- **Maintenance**
- **Disable**

The **Coordinates** section expresses the exact location of the element in pixels, and as the previous item, you can configure some dedicated URL for this kind of host also.

Imagine that you have produced different kinds of screens (the screens will be discussed later in this chapter), one that collects all the metrics graphs and triggers used to monitor an application server, and another one with all the metrics needed to monitor the status of your database. Well, here if your host is a DBMS, you can create a URL to jump directly on the RDBMS screen. If it is an application server, you can create a custom URL that will drive you directly to the application server screens, and so on.

As you can see, this is an interesting feature and will make your map useful for your support team.

Selecting elements

In the map configuration, you can select multiple elements by selecting the first one, and then keeping the *Crtl* (or *Shift*) key pressed, selecting the other elements. For a multiple selection, you can drag a rectangle area which will then select all the elements in the drawn rectangle.

Once you have selected more than one element, the element form switches to the **Mass update elements** as shown in the following screenshot:

Mass update elements (2 elements)

Selected elements:

Host	HOST-F
Host	HOST-G

☐ Label

☐ Label location - ▾

☐ Automatic icon selection ☐

☐ Icon (default) Cloud_(128) ▾

☐ Icon (problem) Default ▾

☐ Icon (maintenance) Default ▾

☐ Icon (disabled) Default ▾

Apply Remove Close

Here you can update the icons, label, and label location for all the selected hosts in bulk.

To have an efficient update of all the labels, it is strongly advised to use macros.

Now it's time to link your server between them, exactly the way they are physically connected. To create a link between two hosts, you only need to select the couple of hosts that need to be linked together, and click on the + symbol in the **Link** section of the map.

The links section will appear right below the **Mass update elements** form, as shown in the following screenshot:

Links between the selected elements			
	From	To	Link indicators
Edit	HOST-F	HOST-G	

Label

Type (OK) Line

Colour (OK) 00CC00

Link indicators

Triggers	Type	Colour
Add		

Apply Remove Close

You can customize your link with labels and also change the representation type and color. You can choose between a **Line**, **Bold line**, **Dot**, and **Dashed line**.

An option to keep in mind here is the possibility to connect the link indicator to a trigger, so basically the link will change its color once a trigger is on fire.



Here you can connect the link to multiple triggers and associate a different color to each one of them so that you can understand which trigger is changing your link.

Playing with macros inside maps

Previously, we discussed about the **Label** section where you can customize the label in your graphs. Here I think an example can clarify a lot about the power of this section, and how this can improve and introduce benefits in your maps. As an example, you can play with macros inside the map. Now you have some requirement for this, such as, you need to show directly in the map the hostname, IP address, the status of triggers events (how many acknowledged and how many unacknowledged), and the network traffic of your network interfaces.

This seems as a challenging work, and in fact it is; but if you have a bit of knowledge about macros, this becomes an easy task. The request can be satisfied with the following code:

```
{HOSTNAME}  
{HOST.CONN}  
trigger events ack: {TRIGGER.EVENTS.ACK}  
trigger events unack: {TRIGGER.EVENTS.UNACK}  
Incoming traffic: {{HOSTNAME}:net.if.in[eth0].last(0)}  
Outgoing traffic: {{HOSTNAME}:net.if.out[eth0].last(0)}
```

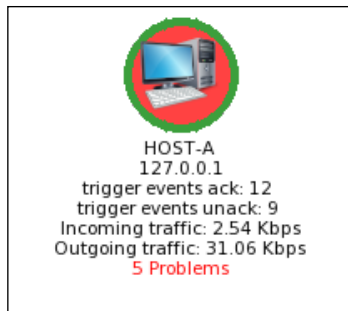
The first macro `{HOSTNAME}` will display the hostname of your selected host. The second macro `{HOST.CONN}` will display the IP address. The information about the trigger's events, whether being acknowledged or unacknowledged is expressed in the next two lines using the macros `{TRIGGER.EVENTS.ACK}` and `{TRIGGER.EVENTS.UNACK}`. The last two lines are more interesting because they are a composition of two nested macros.


In particular, to display the incoming traffic of your first network interface, you can ask Zabbix to retrieve the last value of the `net.if.in[eth0]` item. This kind of expression needs the hostname to be evaluated, so you need to write your hostname, for example, `HOST-A` in this example, or use macros.

The last information that Zabbix needs to produce as our requested output is the hostname. As mentioned earlier, this can be replaced with the `{HOSTNAME}` macro. So the complete expression will be as follows:

```
Incoming traffic: {{HOSTNAME}:net.if.in[eth0].last(0)}
```

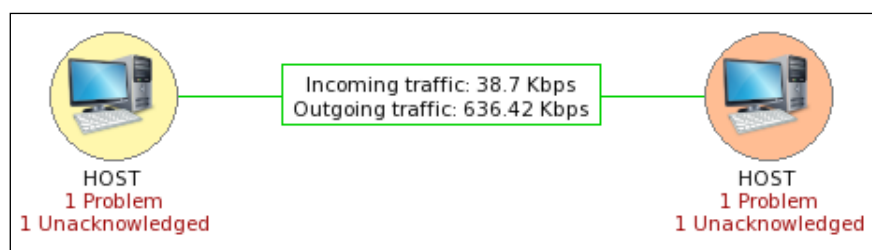
Obviously for the outgoing traffic, the expression is more or less the same, except that you need to retrieve the `net.if.out[eth0]` item of the network card. The result is shown in the following screenshot:



[ Use {HOSTNAME} or {HOST.NAME} in all the labels and all the places where it is possible, so it will make things easy in case of a mass update. **]**

This is a comprehensive and charming output, wherein without any clicks, you have your needed information directly in your map. In this example, you used the `last()` value of your item, but the other function are also supported here such as `last()`, `min()`, `max()`, and `avg()`.

Macros can be used in the same manner on links; an example is shown in the following screenshot:



In the preceding screenshot the traffic data on the link is generated using the same method previously explained. All those macro usages make your maps a lot more dynamic and appealing.

Visualizing through screens

In the previous section, we discussed about adding custom URLs and introduced shortcuts to some screen section. Now it's time to go deep into **screens**. Screens are easy to generate, and very intuitive to handle. Basically, a screen is a *page* that can display multiple Zabbix elements, such as graphs, maps, and text. One of the main difference between screens and maps is that in maps, you can place a lot of elements, but you can't for instance, add a graph or the trigger status. They have two different targets. The screen can group all the elements that are common to a particular kind of server to have a complete picture of the situation.

Creating a screen

To create a screen, you need to navigate to **Configuration | Screen | Create**. A form will appear, asking for the screen name and the initial size in terms of columns and rows. After this step, you need to come back inside the screen that you just created.

In this part of the configuration, probably you will notice that there isn't a save button. This is because screens are saved automatically every time that you complete an action such as add a graph. The screen appears like a table (basically it is a table), as shown in the following screenshot:

CONFIGURATION OF SCREEN			
First screen			

Now if you need more rows or column, you only need to click on the + sign where you want to add fields as rows or as column.

On a screen, you can add the following kinds of elements:

- Data overview: This displays the latest data for a group of hosts
- Graph: This displays whether the graphs are single or custom
- History of actions: This displays n lines of recent actions
- History of events: This displays n lines (you can specify how many) of latest events
- Hosts info: This displays a high-level host related information
- Map: This displays a single map
- Plain text: This shows the plain text data
- Screen: This displays another screen (one screen may contain other screens inside it)
- Server info: This displays the high-level server information
- Status of hostgroup triggers: This displays the status of triggers filtered per hostgroup
- Status of host triggers: This displays the status of triggers filtered per host
- System status: This displays the system status (it is close to the **Dashboard**)
- Triggers info: This displays the high-level trigger related information
- Triggers overview: This displays the status of triggers for a selected host group
- URL: Here you can include content from an external source

All those sources have two common configuration parameters – the column span and the row span. With the column span you can extend a cell to a certain number of columns. So also, with the row span, you can extend a cell for a certain number of rows. For instance, in a table of two columns, if you indicate a column span of two, the cell will be centered in the table and will be widened with exactly two fields. This is useful to add a header to your screen.

Once you have inserted and configured your elements on the screen, you can move it using drag-and-drop, and all your settings will be preserved.



You can freely drag-and-drop your configured element; they will not lose their settings.

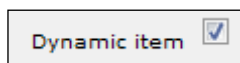
Most of the elements are not dynamic, which means that they will not be dynamically applied to all your hosts in a group.

Dynamic elements

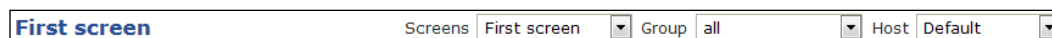
Zabbix provides some dynamic elements that are really useful, which are as follows:

- Graphs
- Simple graphs
- Plain text

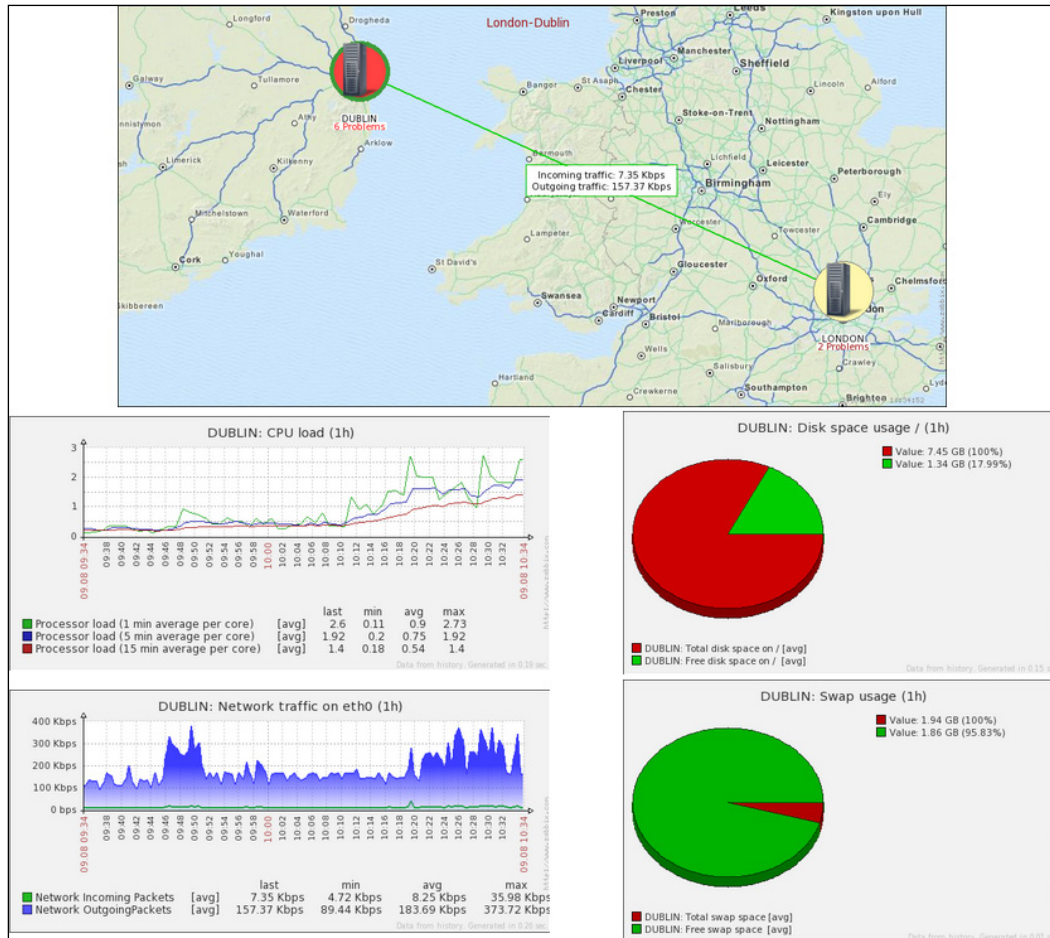
Dynamic items can be identified by having the following option checked:



Now for instance, you can add a map in your screen and some dynamic elements such as graphs. When you add a dynamic element on top of the screen, you will have a bar with some listboxes that will help you to change the target of your dynamic elements, as shown in the following screenshot:



An example of a screen that mixes dynamic elements and standard elements is shown in the following screenshot:



Obviously, when you choose and host this, it will affect only the dynamic graphs. You can switch between your two hosts and change the x axis. This will update all the dynamic elements on the same time base, making it easy to relate elements between them.



Pie graphs and exploded pie graphs will display only the average value for your chosen period. To relate different groups of metrics between them, it is better to use a custom graph.

Visualizing the data through a slide show

Once you have created all your screens, you can provide a slide show to your helpdesk that implements a screen rotation.

Creating a slide show is easy; go to **Configuration | Slide shows | Create slide show**. A window as shown in the following screenshot will appear:

Slide																	
Name	First slide show																
Default delay (in seconds)	30																
Slides	<table border="1"> <thead> <tr> <th></th> <th>Screen</th> <th>Delay</th> <th>Action</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>First screen</td> <td>default</td> <td>Remove</td> </tr> <tr> <td>2</td> <td>General screen</td> <td>default</td> <td>Remove</td> </tr> <tr> <td>3</td> <td>Zabbix server</td> <td>default</td> <td>Remove</td> </tr> </tbody> </table>		Screen	Delay	Action	1	First screen	default	Remove	2	General screen	default	Remove	3	Zabbix server	default	Remove
	Screen	Delay	Action														
1	First screen	default	Remove														
2	General screen	default	Remove														
3	Zabbix server	default	Remove														
	Add																

You can see the slide show configuration in the preceding screenshot. This configuration screen is really intuitive; **Name** identifies the name of your slide show, and in **Default delay (in seconds)** you need to specify the default delay that will be applied to all the screens on the slide show.

In the **Slides** section—enumerated in the preceding screenshot to have a visualization order—you can specify a different delay for each one of them. Basically you can customize how long each screen will be displayed. Once saved, your slide show will be available to be displayed by going to **Monitoring | Screens**, and then select the slide show you just created.



On slide show, you can only create a screen rotation. So to add a single element such as a graph or a map on your rotation, you need to create a screen that contains your elements to be displayed. By using this method, you can basically add all that can be represented on a screen on the slide show.

If you want to speed up or slow down your slide show, you can change the refresh multiplier that will appear by clicking on the **Menu** icon (on the right-hand side of the listbox), which will return a window as shown in the following screenshot:

Refresh time multiplier
x0.25
x0.5
x1
x1.5
x2
x3
x4
x5

Controlling centre slides and the big display challenge

Displaying data on a big display is something challenging; first of all you need to know who will be your target, their skills, and which role do they exactly cover. After that, it is important to see where the screen is physically and the resolution of the same.

You will most probably need to create an ad hoc screen for a big display to fit better on a widescreen. Screens for widescreen need to be developed horizontally. Most of the screens are usually developed with a web page in mind, so most probably they need to be scrolled down and up to read all the graphs. Such kinds of screens will not fit on your widescreen.

You need to bear in mind that the slide show doesn't scroll up and down automatically. You can add a JavaScript to make it possible, but it is really complex to implement a screen that will rightly handle this scroll up and down, and all this effort can be hardly justified. It is better and more productive to produce screens, so that slide shows fit in the screen dimensions and resolution.

Some consideration about slides on a big display

Once you have understood which one your target is, their knowledge, and typology of the work they do, you are already in a stable position. Now you can apply some best practice which is generally useful when you need to show data on a big display. Basically they need to be one of the following:

- Easy to read (comprehensive)
- Fit the big screen display
- Non-interactive
- Choose an appropriate delay between the screens

First of all keep things easy. This is a general rule: the easier are the representations, the lesser the effort required by the helpdesk to read them. An easy and comprehensive screen will improve the helpdesk's reaction, which is our goal. You need to provide information in the best way. Never overload your screen with information; you need to choose the right amount of information to be displayed and fonts need to be comprehensive. Essentially you need to choose which cut will have every screen, and the challenge is to choose how to display your monitored services.

If you need to monitor a large amount of services, you need to choose the time to change the slide; don't spend too much time on the same screen. Keeping a screen for too much time on the monitor can become annoying, especially when the helpdesk is waiting to see the next slide. Unfortunately there isn't a rule; you need to spend time with the first-level support and check with them as to what is the perfect timing.

One thing that simplifies the work is that you don't need to think about complex screens. A widescreen display doesn't need the drill-down feature implemented. People will just look at the screen, and the analysis will be done from their workstation.

Triggers are always welcome as they are easy to read, comprehensive, and immediate. But take care not to fill a page with them as it will then be unreadable.

Automated slide show

Once your slides are created, and you are ready to run them, it is time to think about the user. Your widescreen and the relative workstation dedicated to this task needs to have an account for sure.

In a real-world scenario, we do not want to see the login page on a big display. To avoid this, it is better to create a dedicated account with some customization.

The requirements are as follows:

- Avoid an automatic disconnection
- Avoid the clicks needed to display the slide show

Both these features will be appreciated from your helpdesk.

To avoid the automatic disconnection, there is a flag on the user configuration form that is designed for that—the **Auto-login flag**. Once selected, you need to log in only the first time.



The Auto-login flag will work only if your browser supports cookies; please ensure that plugins, antivirus, and so on are not blocked.

Now since you have created a dedicated account, you need to customize the URL (after login) section; here you need to link the URL to your screen.

To retrieve the appropriate URL, browse to your slide show and copy your link. For this example, the link would be as follows:

```
http://<your-zabbix-server>/zabbix/slides.  
php?sid=4258s278fa96eb&form_refresh=1&fullscreen=0&elementid=3
```

Basically, you need to know the `elementid` value of your slide show. In the following URL, you can remove the `sid` parameter. The definitive URL in our case will be as follows:

```
http://<your-zabbix-server>/zabbix/slides.php? form_refresh=1&fullscr  
een=0&elementid=3
```







To jump directly to the full screen mode, change the parameter `fullscreen=0` to `fullscreen=1`. This will further reduce the human interaction.

Now this account has a first page. After login, the slide show starts in the full screen mode with really little human interaction.

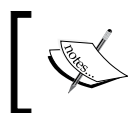
IT services

The last graphical element that will be discussed in this chapter is a high-level view of our monitored infrastructure. In a business-level view, there is no provision for low-level details such as the CPU usage memory consumption and free space. What the business would like to see is the availability of your services provided, and the service-level agreements of your IT services.

Zabbix covers this point with **IT services**. A service is a hierarchical view of your service. Now imagine that you need to monitor your website (we discussed about SLAs in *Chapter 1, Deploying Zabbix*). You need to identify your service components for example, Web server, Application server, and DB server. For each one of them you need to identify triggers that tell you whether the service is available or not. The hierarchical view is the one represented in the following screenshot:

Service	Status	Reason	Problem time	SLA / Acceptable SLA
root				
WEBSITE SLA Calculated	OK	-	 0.0000	100.0000/ 99.9000
Web - Service on WEBSERVER is unavailable	OK	-	 0.0000	100.0000/ 99.9000
RDBMS - Service on DATABASE SERVER is unavailable	OK	-	 0.0000	100.0000/ 99.9000
Application - Service on DATABASE SERVER is unavailable	OK	-	 0.0000	100.0000/ 99.9000

In this hierarchy each node has a status; this status is calculated on trigger basis, and propagated to the higher level with the selected algorithm. So the lowest level of IT services is managed via triggers.



Triggers are the core of IT service calculations; so as you can imagine, they are of particular importance and really critical. You need to find out which are your effective items to check for this trigger generation.

Triggers with severity as Information and Not classified are not considered, and don't affect the SLA calculation.

Configuring an IT service

The way to configure an IT service is by navigating to **Configuration | IT services**; you can create your service here. The following screenshot displays the service previously configured:

Service	Status calculation	Trigger
root		
[-] WEBSITE SLA Calculated	Problem, if at least one child has a problem	-
[-] Web	least one child has a problem	Service on WEBSERVER is unavailable
[-] Add service	least one child has a problem	Service on DATABASE SERVER is unavailable
[-] Edit service	least one child has a problem	Service on DATABASE SERVER is unavailable
[-] Delete service		

By clicking on a service you can add a service, edit the current service, or delete it. The service configuration is composed of three forms: the first one describes the service, the second tab is dedicated to the dependencies, and the third is to the time.

In the service tab, you need to define your own service name. In this particular example, the website SLA is calculated; of course a website is composed of different components such as the Web server, Application server, and DBMS. In a three-level environment, they are usually on a dedicated server. Now since all the three components are vital for our merchant website, we need to calculate the SLA propagating the problems. This means that if the child of our website has a problem, the whole website has a problem, and this will reflect in the SLA calculation.

Zabbix provides the following three options on the status calculation algorithm:

- **Do not calculate:** This option ignores the status calculation completely.
- **Problem, if at least one child has a problem:** This means that if each one of our three components has an issue, the service will be considered unavailable. This is the case when each one of the server doesn't have a failover node.
- **Problem, if all the children has a problem:** To propagate the problem, all the children need to be affected by the problem. This case is typical for a clustered or load-balanced service, where there are many nodes to provide a service, and all the nodes need to be down to propagate the issue to the parent node.

Once you define the algorithm, you need to define the SLA percentage of your service. This is used to display the SLA issue with different colors on the report.

The next step is the trigger definition that will enable Zabbix to know when your service has an issue. Since Zabbix provides a hierarchical view, you can have a service composed of many components so the intermediate level can avoid a trigger definition that is needed on the lowest level.

The last option is **Sort order(0->999)**. This of course doesn't affect the SLA calculation, but is only for cosmetic purposes. To visualize on report, for instance, your three levels are sorted in a logical order as Web server, Application server, and DB server. All that is previously discussed is shown in the following screenshot:

The screenshot shows the 'Time' tab of the Zabbix Service configuration page. The service name is 'WEBSITE SLA Calculated' and its parent is 'root'. The status calculation is set to 'Problem, if at least one child has a problem'. The 'Calculate SLA' checkbox is checked, with a value of '99.9000'. The 'Trigger' field is empty, and the 'Sort order (0->999)' is set to '0'.

The following screenshot shows the dependencies; here you don't need to define each one of them, because they are defined automatically once you design your hierarchical view. Now it is possible that one of your service is already defined for some reason in another layer of the service. If this is the case, you only need to mark the service as soft linked, by ticking the **Soft** checkbox.


The screenshot shows the 'Dependencies' tab of the Zabbix Service configuration page. It displays a table of dependencies for the service 'WEBSITE SLA Calculated'. The table has columns for 'Services', 'Soft', 'Trigger', and 'Action'. The dependencies listed are 'Web', 'RDBMS', and 'Application', all of which are marked as 'Soft' (checkbox checked). The trigger for each is 'Service on WEBSERVER is unavailable', 'Service on DATABASE SERVER is unavailable', and 'Service on DATABASE SERVER is unavailable' respectively. The action for each is 'Remove'.



If a service has only soft linked dependencies, it can be deleted. In this case, you don't need to delete all the children services first; this can be used to quickly delete the whole service.

The last tab is used to set the service time. By default, Zabbix considers that a service needs to be available 24 hours a day, for seven days, and the whole year (24 x 7 x 365). Fortunately for the system administrators, not all the services have this requirement. If you are on this case, you can define your **Uptime** and **Downtime** periods, as shown in the following screenshot:

The screenshot shows the 'Time' tab in the Zabbix service configuration interface. It features a table for 'Service times' with columns 'Type', 'Interval', 'Note', and 'Action'. Below the table is a 'New service time' section with a dropdown menu set to 'Uptime'. There are also fields for 'From' (Sunday), 'Till' (Sunday), and 'Time' (hh:mm). An 'Add' button is located at the bottom of the form.

[ The periods defined here are basically **Uptime** and **Downtime** windows. A problem that occurs during a **Downtime** window will not affect the SLA calculation. Here it is possible to also add a one-time downtime, useful to define an agreed maintenance without an impact on the SLA.]

Once you completed the hierarchical definition of your service, the result is available by navigating to **Monitoring | IT services**.

Summary

In this chapter, we covered the entire graphical elements that Zabbix provides, and showed how to use them in an efficient way. This chapter also enabled you to deliver efficient slide shows to your helpdesk, making you aware of the best practices in this difficult task. Now you probably understood that this part of the Zabbix setup will require time to be well implemented. Also, it is easier to understand for the masses, and the information provided with the graphical elements has a big impact on your audience. This forces you to be precise and take a lot of care in this task, but this will be paid back to you providing a lot of powerful elements to use, to provide more strength to your speech. All those graphical elements are fundamentals if you need to argue with the business or the purchase manager to expand the business or buy hardware.

In the next chapter, you will see how to manage complex triggers and trigger conditions. The next chapter will also make you aware of the right amount of triggers and alarms you should implement to not be overloaded by alarms, with the consequence of losing the critical ones.

6

Managing Alerts

Checking conditions and alarms are the most characteristic functions of any monitoring system, and Zabbix is no exception. What really sets Zabbix apart is that every alarm condition, or triggers – as they are known in this system – can be tied, not only to a single measurement but also to an arbitrary complex calculation, based on all of the data available to the Zabbix server. Furthermore, just as triggers are independent from items, the actions that the server can take based on trigger status are independent from the single trigger, as you will see in the subsequent sections.

In this chapter, you will learn the following things about triggers and actions:

- How to create complex, intelligent triggers
- How to minimize the possibility of false positives
- How to set up Zabbix to take automatic actions based on the trigger status
- How to rely on escalating actions


An efficient, correct, and comprehensive alerting configuration is a key to the success of a monitoring system. It's based on an extensive data collection, as discussed in *Chapter 4, Collecting Data*, and eventually leads to managing messages, recipients, and delivery media, as we'll see later in the chapter. But all this revolves around the conditions defined for the checks, and this is the main business of triggers.

Understanding trigger expressions

Triggers are quite simple to create and configure—choose a name and a severity, define a simple expression using the expression form, and you are done. The expression form, accessible through the **Add** button, lets you choose an item, a function to perform on the item's data, and some additional parameters, and gives an output as shown in the following screenshot:

The screenshot shows a configuration window for a trigger. The 'Dependencies' tab is active. The 'Name' field is set to 'test'. The 'Expression' field contains the expression `{Alpha:vm.memory.size[available].delta(600)}>1000`. An 'Add' button is located to the right of the expression field. Below the expression field is a link labeled 'Expression constructor'. At the bottom of the window, there is a checkbox labeled 'Multiple PROBLEM' and a label 'events generation'.

You can see how there's a complete item key specification, not just the name, to which a function is applied. The result is then compared to a constant using the *greater than* operator. The syntax for referencing the item keys is very similar to that of a calculated item. In addition to this basic way of referring to item values, triggers also add a comparison operator that wraps all the calculations up to a Boolean expression. This is the one great unifier of all triggers; no matter how complex the expression, it must always return either a `True` value or a `False` value. This value is of course directly related to the state of a trigger, which can only be `OK` if the expression evaluates to `False`, or `PROBLEM`, if the expression evaluates to `True`. There are no intermediate or soft states for triggers.

 A trigger can also be in an `UNKNOWN` state if it's impossible to evaluate the trigger expression (because one of the items has no data, for example).

A trigger expression has two main components:

- Functions applied to the item data
- Arithmetical and logical operations performed on the functions' results

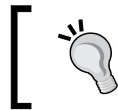
From a syntactical point of view, the *item and function* component has to be enclosed in curly brackets, as illustrated in the preceding screenshot, while the arithmetical and logical operators stay outside the brackets.

Selecting items and functions

You can reference as many items as you want in a trigger expression, as long as you apply a single function to every single item. This means that if you want to use the same item twice, you'll need to specify it twice completely, as shown in the following code:

```
{Alpha:log[/tmp/operations.log,,10,skip].nodata(600)}=1 |
{Alpha:log[/tmp/operations.log,,10,skip].str(error)}=1
```

The previously discussed trigger will evaluate to **PROBLEM** if there are no new lines in the `operations.log` file for more than ten minutes, or if an error string is found in the lines appended to that same file.



Zabbix doesn't apply short-circuit evaluation of the *and* and *or* (& and |) operators—every comparison will be evaluated regardless of the outcome of the preceding ones.

Of course, you don't have to reference items from the same host; you can reference different items from different hosts, and on different nodes too, if you can access them, as shown in the following code:

```
{Node1:Alpha:agent.ping.last(0)}=0 &
{Node2:Beta:agent.ping.last(0)}=0
```

Here, the trigger will evaluate to **PROBLEM** if both the hosts Alpha and Beta are unreachable. It doesn't matter that the two hosts are monitored by two different nodes. Everything will work as expected as long as the node where the trigger is defined has access to the two monitored hosts' historical data. In other words, the trigger has to be configured on a master node that receives data from both Node1 and Node2.

You can apply all the same functions available for calculated items to your items' data. The complete list and specification is available on the official Zabbix wiki (<https://www.zabbix.com/documentation/2.0/manual/appendix/triggers/functions>), so it would be redundant to repeat them here, but a few common aspects among them deserve a closer look.

Choosing between seconds or number of measurements


Many trigger functions take a *sec* or *#num* argument. This means that you can either specify a time period in seconds, or a number of measurements, and the trigger will take all of the item's data in the said period, and apply the function to it. So, the following code will take the minimum value of Alpha's CPU idle time in the last ten minutes:

```
{Alpha:system.cpu.util[,idle].min(600)}
```

The following code, unlike the previous one, will perform the same operation on the last ten measurements:

```
{Alpha:system.cpu.util[,idle].min(#10)}
```

Will perform the same operation on the last ten measurements.

 Instead of a value in seconds, you can also specify things such as 10m for ten minutes, 2d for two days, and 6h for six hours.

Which one should you use in your triggers? While it obviously depends on your specific needs and objectives, each one has its strengths that makes it useful in the right context. For all kinds of passive checks initiated by the server, you'll often want to stick to a time period expressed as an absolute value. A #5 parameter will vary quite dramatically as a time period if you vary the check interval of the relative item. It's not usually obvious that such a change will also affect related triggers. Moreover, a time period expressed in seconds may be closer to what you really mean to check, and thus may be more easy to understand when you visit the trigger definition at a later date. On the other hand, you'll often want to opt for the #num version of the parameter for many active checks, where there's no guarantee that you will have a constant, reliable interval between measurements. This is especially true for trapper items of any kind, and for logfiles. With these kinds of items, referencing the number of measurements is often the best option.

Date and time functions

All the functions that return a time value, whether it's the current date, the current time, the day of the month, or the day of the week, still need a valid item as part of the expression. These can be useful to create triggers that may change status only during certain times of day, or during certain specific days, or better yet, to define well known exceptions to common triggers, when we know that some otherwise unusual behavior is to be expected. For example, a case where there's a bug in one of your company's applications that causes a rogue process to quickly fill up a filesystem with huge logfiles. While the development team is working on it, they ask you to keep an eye on the said filesystem, and kill the process if it's filling the disk up too quickly. Like with many things in Zabbix, there's more than one way to approach this problem, but you decide to keep it simple and find that, after watching the trending data on the host's disk usage, a good indicator that the process is going rogue is that the filesystem has grown by more than 3 percent in 10 minutes:

```
{Alpha:vfs.fs.size[/var,pused].delta(600)}>3
```

The only problem with this expression is that there's a completely unrelated process that makes a couple of big file transfers to this same filesystem every night at 2 a.m. While this is a perfectly normal operation, it could still make the trigger switch to a `PROBLEM` state and send an alert. Adding a couple of time functions will take care of that, as shown in the following code:

```
{Alpha:vfs.fs.size[/var,pused].delta(600)}>3 &  
( {Alpha:vfs.fs.size[/var,pused].time(0)}<020000 |  
  {Alpha:vfs.fs.size[/var,pused].time(0)}>030000 )
```

Just keep in mind that all the trigger functions return a numerical value, including the date and time ones, so it's not really practical to express fancy dates like "the first Tuesday of the month" or "last month" (instead of the last 30 days).

Trigger severity

Severity is little more than a simple label that you attach to a trigger. The web frontend will display different severity values with different colors, and you will be able to create different actions based on them, but they have no further meaning or function in the system. This means that the severity of a trigger will not change over time based on how long that trigger has been in a `PROBLEM` state, nor can you assign a different severity to different thresholds in the same trigger. If you really need a warning alert when a disk is over 90 percent full, and a critical alert when it's 100 percent full, you will need to create two different triggers with two different thresholds and severity. This may not be the best course of action though, as it could lead to either warnings that are ignored and not acted upon, or critical warnings that will fire up when it's already too late and you have already lost service availability, or even just a redundant configuration with redundant messages and more possibilities for mistakes, or an increased signal-to-noise ratio.

A better approach would be to clearly assess the actual severity of the possibility for the disk to fill up, and create just one trigger with a sensible threshold, and possibly an escalating action if you fear that the warning could get lost among the others.

Choosing between absolute values and percentages

If you look at many native agent items, you'll see that a lot of them can express measurements either as absolute values or as percentages. It often makes sense to do the same while creating one's own custom items, as both representations can be quite useful in and on themselves. When it comes to creating triggers on them though, the two can differ quite a lot, especially if you have the task to keep track of available disk space.

Filesystems' sizes and disk usage patterns vary quite a lot between different servers, installations, application implementations, and user engagement. While a free space of 5 percent of a hypothetical disk A could be small enough that it would make sense to trigger a warning and act upon it, the same 5 percent could mean a lot more space for a large disk array, enough that you don't really need to act immediately, but can plan a possible expansion without any urgency. This may lead you to think that percentages are not really useful in these cases, and even that you can't really put disk-space-related triggers in templates, as it would be better to evaluate every single case and build triggers that are tailor-made for every particular disk with its particular usage pattern. While this can certainly be a sensible course of action for particularly sensible and critical filesystems, it can quickly become too much work in a large environment where you may need to monitor hundreds of different filesystems.

This is where the delta function can help you create triggers that are general enough that you can apply them to a wide variety of filesystems, so that you can still get a sensible warning about each one of them. You will still need to create more specialized triggers for those special, critical disks, but you'd have to anyway.

While it's true that the same percentages may mean quite a different thing for disks with a great difference in size, a similar percentage variation of available space on a different disk could mean quite the same thing; the disk is filling up at a rate that can soon become a problem:

```
{Template_fs:vfs.fs.size[/,pfree].last(0)}<5 &
({Template_fs:vfs.fs.size[/,pfree].delta(1d)} /
{Template_fs:vfs.fs.size[/,pfree].last(0,1d)} > 0.5)
```

The previously discussed trigger would report a `PROBLEM` state not just if the available space is less than 5 percent on a particular disk, but also if the available space has been reduced by more than half in the last 24 hours (don't miss the time-shift parameter in the last function). This means that no matter how big the disk is, based on its usage pattern, it could quickly fill up very soon. Note also how the trigger would need progressively smaller and smaller percentages for it to assume a `PROBLEM` state, so you'd automatically get more frequent and urgent notifications as the disk is filling up.

For these kinds of checks, percentage values should prove more flexible and easy to understand than absolute ones, so that's what you probably want to use as a baseline for templates. On the other hand, absolute values may be your best option if you want to create a very specific trigger for a very specific filesystem.

Understanding operations as correlations

As you may have already realized, practically every interesting trigger expression is built as a logical operation between two or more simpler expressions. Naturally, it is not that this is the only way to create useful triggers. Many simple checks on the status of an `agent.ping` item can literally save the day when quickly acted upon, but Zabbix also makes it possible, and relatively easy, to define powerful checks that would require a lot of custom coding to implement in other systems. Let's see a few more examples of relatively complex triggers.

Going back to the date and time functions, let's say that you have a trigger that monitors the number of active sessions in an application, and fires up an alert if that number drops too low during certain hours, because you know that there should always be a few automated processes creating and using sessions in that window of time (from 10:30 to 12:30 in this example). During the rest of the day, the number of sessions is neither predictable nor that significant, so you keep sampling it but don't want to receive any alert. A first, simple version of your trigger could look like the following code:

```
{Appserver:sessions.active[myapp].min(300)}<5 &
{Appserver:sessions.active[myapp].time(0)} > 103000 &
{Appserver:sessions.active[myapp].time(0)} < 123000
```



The `session.active` item key could reference a custom script, a calculated item or anything else. It's used here as a label to make the example easier to read, and not as an instance of an actual ready-to-use native item.



The only problem with this trigger is that if the number of sessions drops below five in that window of time, but it doesn't come up again until after 12:30, the trigger will stay in the `PROBLEM` state until the next day. This may be a great nuisance if you have set up multiple actions and escalations on that trigger, as they would go on for a whole day no matter what you do to address the actual sessions problems. But even if you don't have escalating actions, you may have to give accurate reports on these event durations, and an event that looks like it's going on for almost 24 hours would be both incorrect in itself and for any SLA reporting. Even if you don't have reporting concerns, displaying a `PROBLEM` state when it's not there anymore is a kind of false positive that will not let your monitoring team focus on the real problems, and over the time, may reduce their attention on that particular trigger.

A possible solution is to make the trigger return to an `OK` state outside the target hours, if it was in a `PROBLEM` state, as shown in the following code:

```
(( {Appserver:sessions.active[myapp].min(300)}<5 &
{Appserver:sessions.active[myapp].time(0)} > 103000 &
{Appserver:sessions.active[myapp].time(0)} < 123000)) |
( {TRIGGER.VALUE}=1 &
{Appserver:sessions.active[myapp].min(300)}<0 &
({Appserver:sessions.active[myapp].time(0)} < 103000 |
{Appserver:sessions.active[myapp].time(0)} > 123000) )
```

The first three lines are identical to the trigger defined before. This time there is one more complex condition, as follows:

- The trigger is in a `PROBLEM` state (see the note about the `TRIGGER.VALUE` macro)
- The number of sessions is less than zero (this can never be true)
- We are outside the target hours (the last two lines are the opposite of those defining the time frame preceding it)



The `TRIGGER.VALUE` macro represents the current value of the trigger expressed as a number. A value of 0 means OK, 1 means `PROBLEM`, and 2 means `UNKNOWN`. The macro can be used anywhere you could use an `item.function` pair, so you'll typically enclose it in curly brackets. As you've seen in the preceding example, it can be quite useful when you need to define different thresholds and conditions depending on the trigger's status itself.

The condition about the number of sessions being less than zero makes sure that outside the target hours, if the trigger was in a `PROBLEM` state, the whole expression will evaluate to false anyway. False means the trigger is switching to an OK state.

Here, you have not only made a correlation between an item value and a window of time to generate an event, but you have also made sure that the event will always spin down gracefully instead of potentially going out of control.

Another interesting way to build a trigger is to combine different items from the same hosts, or even different items from different hosts. This is often used to spot incongruities in your systems' state that would otherwise be very difficult to identify.

An obvious case could be that of a server that serves content over the network. Its overall performance parameters may vary a lot depending on a great number of factors, and so it would be very difficult to identify sensible trigger thresholds that wouldn't generate a lot of false positives, or even worse, missed events. What may be certain though is that if you see a high CPU load while network traffic is low, then you may have a problem, as shown in the following code:

```
{Alpha:system.cpu.load[all,avg5].last(0)} > 5 &
{Alpha:net.if.total[eth0].avg(300)} < 1000000
```


An even better example would be the necessity to check for hanging or frozen sessions in an application. The actual way to do it would depend a lot on the specific implementation of the said application, but for illustrative purposes, let's say that a frontend component keeps a number of temporary session files in a specific directory, while the database component populates a table with the session data. Even if you have created items on two different hosts to keep track of these two sources of data, each number taken alone will certainly be useful for trending analysis and capacity planning, but they need to be compared to check if something's wrong in the application's workflow. Assuming that we have previously defined a local command on the frontend's Zabbix agent that will return the number of files in a specific directory, and an `odbc` item on the database host that will query the DB for the number of active sessions, we could then build a trigger that compares the two values and report a `PROBLEM` state if they don't match:

```
{Frontend:dir.count[/var/sessions].last(0)} #  
{Database:sessions.count.last(0)}
```



The # term in the expression is the *not equal* operator.

Aggregated and calculated items can also be very useful for building effective triggers. The following one will make sure that the ratio between active workers and the available servers doesn't drop too low in a grid or cluster:

```
{ZbxMain:grpsum["grid", "proc.num[listener]", last, 0].last(0)} /  
{ZbxMain:grpsum["grid", "agent.ping", last, 0].last(0)} < 0.5
```

All these examples should help drive home the fact that once you move beyond checking for simple thresholds with single item values, and start correlating different data sources together in order to have more sophisticated and meaningful triggers, there is virtually no end to all the possible variations of trigger expressions that you can come up with.

By identifying the right metrics, as explained in *Chapter 4, Collecting Data*, and combining them in various ways, you can pinpoint very specific aspects of your systems' behavior; you can check logfiles together with the login events and network activity to track down possible security breaches, compare a single server performance with the average server performance in the same group to identify possible problems in service delivery, and much more.

This is, in fact, one of Zabbix's best kept secrets that really deserves more publicity; its triggering system is actually a sophisticated correlation engine that draws its power from a clear and concise method to construct expressions as well as from the availability of a vast collection of both current and historical data. Spending a bit of your time studying it in detail, and coming up with interesting and useful triggers tailor-made for your needs will certainly pay you back tenfold, as you will end up not only with a perfectly efficient and intelligent monitoring system, but also with a much deeper understanding of your environment.

Managing the trigger dependencies

It's quite common that the availability of a service or a host doesn't depend only on the said host in itself, but also on the availability of any other machine that may provide connectivity to it. For example, if a router goes down isolating an entire subnet, you would still get alerts about all the hosts in the said network that will suddenly be seen as unavailable from Zabbix's point of view, even if it's really the router's fault. A dependency relationship between the router and the hosts behind it would help alleviate the problem, because it would make the server skip any trigger check for the hosts in the subnet, should the router become unavailable. While Zabbix doesn't support the kind of host-to-host dependencies that other systems do, it does have a trigger-to-trigger dependency feature that can largely perform the same function. For every trigger definition, you can specify a different trigger upon which your new trigger is dependent. If the parent trigger is in a `PROBLEM` state, the trigger you are defining won't be checked, until the parent returns to an `OK` state. This approach is certainly quite flexible and powerful, but it also has a couple of downsides. The first one is that one single host can have a significant number of triggers, so if you want to define a host-to-host dependency, you'll need to update every single trigger, which may prove to be quite a cumbersome task. You can of course, rely on the mass update feature of the web frontend as a partial workaround. A second problem is that you won't be able to look at a host definition and see that there is a dependency relationship with another host. Short of looking at a host's trigger configuration, there's simply no easy way to display or visualize this kind of relationship in Zabbix.

A distinct advantage of having a trigger-level dependency feature is that you can define dependencies between single services on different hosts. As an example, you could have a database that serves a bunch of web applications on different web servers. If the database is unavailable, none of the related websites will work, so you may want to set up a dependency between the web monitoring triggers and the availability of the database. On the same servers, you may also have some other service that relies on a separate license server, or an identity and authentication server. You could then set up the appropriate dependencies, so that you could end up having some triggers depend on the availability of one server, and other triggers depend on the availability of another one, all in the same host. While this kind of configuration can easily become quite complex and difficult to maintain efficiently, a select few, well-placed dependencies can help cut down the amount of redundant alerts in a large environment. This in turn would help you focus immediately on the real problems where they arise, instead of having to hunt them down in a long list of trigger alerts.

Taking action

Just as items only provide raw data, and triggers are independent from them as they can access virtually any item's historical data, triggers in turn only provide a status change. This change is recorded as an event, just like measurements are recorded as item data. This means that triggers don't provide any reporting functionality; they just check their conditions and change the status accordingly. Once again, what may seem like a limitation and a lack of power turns out to be the exact opposite, as the Zabbix component in charge of actually sending out alerts, or trying to automatically resolve some problems, is completely independent from triggers. This means that just like triggers can access any item's data, actions can access any trigger's name, severity, or status, so that once again you can create the perfect mix of very general and very specific actions, without being stuck in a one-action-per-trigger scheme.

Unlike triggers, actions are also completely independent from hosts and templates too. Every action is always globally defined, and its conditions checked against every single Zabbix event. As you'll see in the following paragraphs, this may force you to create some explicit conditions, instead of implicit conditions, but that's balanced out by the fact that you won't have to create similar but different actions for similar events just because they are related to different hosts.

An action is composed of the following three different parts that work together to provide all the functionality needed:

- Action definition
- Action conditions

- Action operations

The fact that every action has a global scope is reflected in every one of its components, but it assumes a critical importance when it comes to action conditions, as it's the place where you decide which action should be executed based on which events. But let's not get ahead of ourselves and let's see a couple of interesting things about each component in turn.

Defining an action

This is where you decide a name for the action, and can define a default message that can be sent as a part of the action itself. In the message, you can reference specific data about the event, such as the host, item and trigger name, item and trigger values, and URLs. Here, you can leverage the fact that actions are global by using macros, so that a single action definition could be used for every single event in Zabbix, and yet provide useful information in its message.

You can see a few interesting macros already present in the default message when you create a new action, as shown in the following screenshot:

The screenshot shows the Zabbix Action configuration form. It includes fields for Name, Default operation step duration (3600), Default subject, and Default message. The Default message field contains a template with macros for trigger and item data. There is also a section for Recovery message with its own subject and message fields, and an Enabled checkbox at the bottom.

Name:

Default operation step duration: (minimum 60 seconds)

Default subject:

Default message:

Trigger: {TRIGGER.NAME}
Trigger status: {TRIGGER.STATUS}
Trigger severity: {TRIGGER.SEVERITY}
Trigger URL: {TRIGGER.URL}

Item values:
1. {ITEM.NAME1} ({HOST.NAME1}:{ITEM.KEY1}): {ITEM.VALUE1}
2. {ITEM.NAME2} ({HOST.NAME2}:{ITEM.KEY2}): {ITEM.VALUE2}
3. {ITEM.NAME3} ({HOST.NAME3}:{ITEM.KEY3}): {ITEM.VALUE3}

Recovery message: ☒

Recovery subject:

Recovery message:

Trigger URL: {TRIGGER.URL}

Item values:
1. {ITEM.NAME1} ({HOST.NAME1}:{ITEM.KEY1}): {ITEM.VALUE1}
2. {ITEM.NAME2} ({HOST.NAME2}:{ITEM.KEY2}): {ITEM.VALUE2}

Enabled: ☒

Save Cancel

Most of them are pretty self-explanatory, but it's interesting to see how you can of course reference a single trigger, the one that generated the event. On the other hand, as a trigger can check on multiple items from multiple hosts, you can reference all the hosts and items involved (up to nine different hosts and/or items) so that you can get a picture of what's happening by just reading the message.

Other interesting macros can make the message even more useful and expressive. Just remember that the default message may be sent not only via e-mail, but also via a chat program or an SMS; you'll probably want to create different default actions with different messages for different media types, so that you can calibrate the amount of information provided based on the media available.

You can see the complete list of supported macros in the official documentation wiki at https://www.zabbix.com/documentation/2.0/manual/appendix/macros/supported_by_location, so we'll look at just a couple of the most interesting ones.

The {EVENT.DATE} and {EVENT.TIME} macros

These two macros can help differentiate between the time a message is sent and the time of the event itself. It's particularly useful not only for repeated or escalated actions, but also for all media where a timestamp is not immediately apparent.

The {INVENTORY.SERIALNO.A} and friends macros

When it comes to hardware failure, information about a machine's location, admin contact, serial number, and so on, can prove quite useful to track it down quickly, or to pass it on to external support groups.

The {NODE.ID} and {NODE.NAME} macros

These are quite important in a distributed architecture. If you have thousands of monitored hosts managed by different nodes, it may not always be immediately apparent which host is monitored by which node. These macros can really help avoid wasting time just looking for the right node in order to investigate the event.

Defining the action conditions

This part lets you define conditions based on the event's hosts, trigger, and trigger values. Just like trigger expressions, you can combine different simple conditions with a series of AND/OR logical operators, as shown in the following screenshot. Unlike trigger expressions, there is not much flexibility in how you combine them. You can either have all AND, all OR, or a combination of the two, where conditions of different types are combined with AND, while conditions of the same type are combined with OR:

Type of calculation: **AND / OR** (A) and (B) and (C) and (D or E)

Label	Name	Action
(A)	Maintenance status not in "maintenance"	Remove
(B)	Trigger value = "PROBLEM"	Remove
(C)	Trigger severity = "High"	Remove
(D)	Trigger name like "test"	Remove
(E)	Trigger name like "prod"	Remove

New condition: **Trigger name** like

[Save](#)

Zabbix 2.0.6 Copyright © 2006-2015 Zabbix SIA

Observe how one of the conditions is **Trigger value = "PROBLEM"**. Since actions are evaluated for every event, and a trigger switching from PROBLEM to OK is an event in itself, if you don't specify this condition, the action will be executed both when the trigger switches to PROBLEM and when the trigger switches back to OK. Depending on how you have constructed your default message and what operations you intend to do with your actions, this may very well be what you intended, and Zabbix will behave exactly as expected.

Anyway, if you created a different recovery message in the **Action** definition form, and you forget the condition, you'll get two messages when a trigger switches back to `OK`—one will be the standard message, and one will be the recovery message. This can certainly be a nuisance as any recovery message would be effectively duplicated, but things can get ugly if you are relying on external commands as part of the action's operations. If you forget to specify the condition **Trigger value = "PROBLEM"**, the external, remote command would also be executed twice; once when the trigger switches to `PROBLEM` (this is what you intended), and once when it switches back to `OK` (this is quite probably not what you intended). Just to be on the safe side, and if you don't have very specific needs for the action you are configuring, it's probably better if you get into the habit of putting **Trigger value = "PROBLEM"** for every new action you create, or at least check that it's present in the actions you modify.

The most typical application for creating different actions with different conditions is to send alert and recovery messages to different recipients. This is the part where you should remember that actions are global.

Let's say that you want all the database problems sent over to the DB Administrators group and not the default Zabbix Administrators group. If you just create a new action with the condition that the host group must be DB Instances, and as message recipients, choose your DB Admins, what will happen is that they will certainly receive a message for any DB related event, but so will your Zabbix Admins, if the default action has no conditions configured. The reason is that since actions are global, they are always executed whenever their conditions evaluate to `True`. In this case, both the specific action and the default one would evaluate to `True`, so both groups would receive a message. What you could do is add an opposite condition in the default action so that it would be valid for every event, except for those related to the DB Instances host group. The problem is that this approach can quickly get out of control, and you may find yourself with a default action full of the *not in group* conditions. Truth is, once you start creating actions specific for message recipients, you either disable the default action or take advantage of it to populate a message archive for administration and reporting purposes.

Choosing the action operations

If the first two parts were just preparation, this is where you tell the action what it should actually do. The following are two main aspects to this:

- Operation steps
- The actual operations available for each step

As with almost everything else in Zabbix, the simplest cases that are very straightforward are most often self-explanatory; you just have a single step, and this step consists of sending the default message to a group of defined recipients. As with almost everything else in Zabbix, this simple scenario can become increasingly complex and sophisticated, but still manageable, depending on your specific needs. Let's see a few interesting details about each part.

Steps and escalations

Even if an action is tied to a single event, it does not mean that it can perform a single operation. In fact, it can perform an arbitrary number of operations, called steps, which can even go on for an indefinite amount of time, or until the conditions for performing the action are not valid anymore.

You can use multiple steps to both send messages as well as perform some kind of automated operations. Or, you can use the steps to send alert messages to different groups, or even multiple times to the same group, with the time intervals that you want, as long as the event is unacknowledged, or even not yet resolved. The following screenshot shows a combination of different steps:

The screenshot displays the Zabbix Action configuration interface. It is divided into two main sections: 'Action operations' and 'Operation details'.

Action operations: This section shows a table of steps for an action.

Steps	Details	Start in	Duration (sec)	Action
1	Send message to user groups: Guests	Immediately	60	Edit Remove
2	Run remote commands on current host	00:01:00	Default	Edit Remove
3 - 5	Send message to user groups: Zabbix administrators	01:01:00	600	Edit Remove
6	Send message to users: Admin	01:31:00	600	Edit Remove

Operation details: This section provides configuration for a specific step (Step 7).

Step: 7

From:

To: (0 - infinitely)

Step duration: (minimum 60 seconds, 0 - use action default)

Operation type:

Send to User groups:

User group	Action
Add	

Send to Users:

User	Action
Add	

Send only to:

Default message: ☒

Conditions:

Label	Name	Action
(A)	Event acknowledged = "Not Ack"	Remove
New		

[Add](#) [Cancel](#)

As you can see, step 1 starts immediately, and is set to send a message to a user group, and then delays the subsequent step by just one minute. After one minute, step 2 starts and is configured to perform a remote command on the host. As step 2 has a default duration (the duration of which is defined in the main **Action** definition tab), step 3 will start after about an hour. Steps 3, 4, and 5 are all identical and have been configured together – they will send a message to a different user group every 10 minutes. You can't see it in the preceding screenshot, but step 6 will only be executed if the event is not yet acknowledged, just as step 7 that is still being configured. The other interesting bit of step 7 is that it's actually set to configure steps 7 to 0. It may seem against intuition, but in this case, step 0 simply means "forever". You can't really have further steps if you create a step *N* to 0, because the latter will repeat itself with the time interval set in the step's **Duration(sec)** field. Be very careful in using a step 0 because it will really go on until the trigger's status changes. Even then, if you didn't add a **Trigger status="PROBLEM"** condition to your action, a step 0 can be executed even if the trigger switched back to OK. In fact, it's probably best never to use step 0 at all, unless you really know what you are doing.

Messages and media

For every message step, you can choose to send the default message that you configured in the first tab of the **Action** creation form, or send a custom message that you can craft in exactly the same way as the default one. You might want to add more details about the event if you are sending the message via e-mail to a technical group, or reduce the amount of details, or the wording of the message, if you are sending it to a manager or supervisor, or if you are limiting the message to an SMS.

Remember that in the **Action** operation form, you can only choose recipients as Zabbix users and groups, while you still have to specify for every user any media address they are reachable to. This is done in the **Administration** tab of the Zabbix frontend by adding media instances for every single user. You also need to keep in mind that every media channel can be enabled or disabled for a user, or it may be active only during certain hours of the day, or just for one or more specific trigger severity, as shown in the following screenshot:

New media

Type: Email ▼

Send to: admin@site.local

When active: 1-7,00:00-24:00

Use if severity:

- ☒ Not classified
- ☒ Information
- ☒ Warning
- ☒ Average
- ☒ High
- ☐ Disaster

Status: Enabled ▼

Add Cancel

This means that even if you configure an action to send a message, some recipients may still not receive it based on their own media configuration.

While **Email**, **Jabber**, and **SMS** are the default options for sending messages, you still need to specify how Zabbix is supposed to send them. Again, this is done in the **Media types** section of the **Administration** tab of the frontend. You can also create new media types there, which will be made available both in the media section of user configuration, and as targets for message-sending in the **Action** operations form.

A new media type can be a different e-mail, jabber, or an SMS server, in case you have more than one, and you need to use them for different purposes or with different sender identifications. It can also be a script, and this is where things can become interesting if potentially misleading.

A custom media script has to reside on the Zabbix server in the directory indicated by the `AlertScriptPath` variable of `zabbix_server.conf`. When called upon, it will be executed with the following three parameters passed by the server:

- \$1: The recipient of the message
- \$2: The subject of the message
- \$3: The body of the main message

The recipient will be taken from the appropriate user-media property that you would have defined for your users while creating the new media type. The subject and the message body will be the default ones configured for the action, or some step-specific ones, as explained before. Then, from Zabbix's point of view, the script should send the message to the recipient by whatever custom methods you intend to use, whether it's an old UUCP link, or a modern mail server that requires strong authentication, or a post to an internal microblogging server. The fact is, you can actually do what you want with the message; you can simply log it to a directory, send it to a remote file server, morph it to a syslog entry and send it over to a log server, run a speech synthesis program on it and read it aloud on some speakers, or record a message on an answering machine; the sky's the limit with custom media types. This is why you should not confuse a custom media with the execution of a remote command – while you could potentially obtain roughly the same results with one or the other. Custom media scripts and remote commands are really two different things.

Remote commands

These are normally used to try to perform some corrective actions in order to resolve a problem without human intervention. After you've chosen the target host that should execute the command, the Zabbix server will connect to it and ask it to perform it. If you are using the Zabbix agent as a communication channel, you'll need to set `EnableRemoteCommands` to 1, or the agent will refuse to execute any command. Other possibilities are SSH, telnet, or IPMI (if you have compiled the relative options during server installation).

Remote commands can be used to do almost anything – kill or restart a process, make space on a filesystem by zipping or deleting old files, reboot a machine, and so on. They tend to seem powerful and exciting to new implementers, but in the authors' experience, they tend to be fragile solutions that can break things almost as often as they fix it. It's harder than it looks to make them run safely, without accidentally deleting files, or rebooting servers when there's no need to. The real problem with remote commands is that they tend to hide problems instead of revealing them, which should really be the job of a monitoring system. Yes, they can prove useful as a quick patch to ensure the smooth operation of your services, but use them too liberally and you'll quickly forget that there actually are recurring problems that need to be addressed, because some fragile command somewhere is trying to fix things in the background for you. It's usually better to really try to solve a problem than to just hide it behind some automated temporary fix, and not just from a philosophical point of view, but because when these patches fail, they tend to fail spectacularly and with disastrous consequences.

So our advice is to use remote commands very sparingly, and only if you know what you are doing.

Summary

This chapter focused on what is usually considered the "core business" of a monitoring system – its triggering and alerting features. By concentrating separately and alternately on the two parts that contribute to this function: triggers and actions, it should be clear to you how once again Zabbix's philosophy of separating all the different functions can give great rewards to the astute user. You should have learned how to create complex and sophisticated trigger conditions that will help you have a better understanding of your environment, and more control over what alerts you should receive. The various triggering functions and options, as well as some of the finer aspects of item selection, along with the many aspects of action creation should not be a secret to you now.

In the next chapter, you will explore the final part of Zabbix's core monitoring components – the templates and discovery functions.

7

Managing Templates

For all the monitoring powers of Zabbix's items, graphs, maps, and triggers, it would be incredibly cumbersome to manually create every single one of these objects for every monitored host. In the case of a large environment, with hundreds or thousands of monitored objects, it would be practically impossible to manually configure all the items, graphs, and triggers needed for each host.

By using the templates facility, you'll define different collections of items, triggers, and, graphs, in order to apply common configurations to any number of hosts, while still being able to manage any single aspect you may need to tweak for any single host.

The perfect complement to the template facility is Zabbix's discovery feature. By using it, you'll define a set of rules to let Zabbix know of new hosts without having to manually create them. You can also take advantage of the low-level discovery powers of the Zabbix agent, so that you can automatically assign the correct items even for those highly variable parts of a system, such as the number and nature of disks, file systems, and network interfaces.

In this chapter, you will learn to:

- Create and leverage the power of nested templates
- Combine different templates for the same hosts
- Use host discovery and actions to add templates to new hosts
- Configure a low-level discovery to make templates even more general

Let's start from the beginning and see how a template is different from a regular host, even if they look almost the same.

Creating templates

A host template is very similar to a regular host. Both are collections of items, triggers, graphs, screens, and low-level discovery rules. Both need a unique name, just as any other entity in Zabbix. Both can belong to one or more groups. The crucial difference is that a host has one or more means to be contacted so that the Zabbix server can actually take item measurements on it, as illustrated in *Chapter 4, Collecting data*.

These can be one or more IP addresses, or host names, that represent agent interfaces, or SNMP, JMX, and IPMI ones. So a host is an object the Zabbix server will ask for information to, or wait for data from. A template, on the other hand, doesn't have any access interface, so the Zabbix server will never try to check if a template is alive, or ask it for the latest item measurements.

The creation of a template is very straightforward and there is not much to say about it. You navigate to the **Configuration | Templates** tab and click on the **Create template** button. The template creation form that will appear is composed of three different tabs. We'll look at the **Linked templates** tab and the **Macros** tab later in the chapter, as these are not essential to create a basic template. In fact, the only essential element for a basic template is its name, but it can be useful to assign it to one or more groups in order to find it more easily in the other section of the Web interface. If you have configured some hosts already, you can also assign the template to the hosts you're interested in directly from the template creation tab. Otherwise, you'll need to go to the **Hosts** configuration tab and assign templates there. Once you're done, the template is created and available in the template list, but it's still an empty object. Your next job is to create the template's items, trigger, graphs, screens, and discovery rules, if any.

Adding entities to a template

Adding an item or any other entity to a template is virtually identical to the same operation performed on a regular host. This is especially true for items. As you already know, item keys are the basic building blocks of the Zabbix monitoring pipeline and you don't have to specify any kind of address or interface when you create them, as Zabbix will take this kind of information from the host the item is assigned to. This means that when you create items for a template, you are effectively creating items for an ideal host that will be later applied to real ones, once you have linked the template to the hosts you want to monitor.



Templates, just like hosts, are essentially collections of items, triggers, and graphs. Since many of the concepts that we will explore apply equally to items, triggers, and graphs, for the rest of the chapter we'll be using the term entity to refer to any of the three types of objects. In other words, you can understand an item, a trigger, or a graph every time you read entity, and items, triggers, and graphs when you read entities, as a collective term.

This applies to other types of entities as well, but as they always reference one or more existing items, you need to make sure that you select the items belonging to the right template, and not to a regular host:

Items	Node	ZBX Main ▼	Group	Templates ▼	Host	Template App Zabbix Agent ▼
Name	Key	Type	Type of information	Status		
Agent ping	agent.ping	Zabbix agent	Numeric (unsigned)	Enabled		
Host name of zabbix_agentd running	agent.hostname	Zabbix agent	Character	Enabled		
Version of zabbix_agent(d) running	agent.version	Zabbix agent	Character	Enabled		

This may seem obvious but it is far too easy to overlook the **Group** and **Host** selectors at the top of the window when selecting an item and ending with a specific host's item in a template trigger or graph.

The main difference between template entities and host entities, especially when it comes to triggers, is that with template entities the use of macros is quite useful to make trigger and graph names or parameters more expressive and adaptable.

Using macros

As you've already seen in *Chapter 6, Managing Alerts*, macros are very useful to make a message general enough that it can be applied to a wide range of events. It will be the Zabbix server's job to substitute all the macros in a message with the actual content based on the specific event it's handling. Since an action message is effectively a template that has to be applied to a particular event, it's easy to see how the same concept is essential to the effectiveness of host templates. What changes is the context; while an event has a context that is quite rich, since it can reference a trigger and one or more different items and hosts, the context of a simple, regular host is admittedly more limited. This is reflected in the number of macros available, as they are just a handful:

Macro name	Macro translates to	Notes
{HOST.CONN}	Hostname or IP address of the host	Will be identical to either {HOST.IP} or {HOST.DNS} depending on the Connect to option in the host's configuration form.
{HOST.DNS}	The host's hostname	This must correspond to the host's fully qualified domain name as defined in the domain's DNS server.
{HOST.HOST}	The host's name as defined in Zabbix	This is the main host identifier. It must be unique for the specific Zabbix server. If using an agent, the same name must be present in the agent's configuration on the host.
{HOST.IP}	The host's IP address	A host can have more than one IP address. You can reference them using {HOST.IP1}, {HOST.IP2}, and so on up to {HOST.IP9}.
{HOST.NAME}	The host's visible name as defined in Zabbix	This will be the name visible in lists, maps, screens, and so on.

To better clarify the differences between the various {HOST.*} macros, let's see an example host configuration:

The screenshot shows the Zabbix Host configuration page. At the top, there are tabs for Host, Templates, IPMI, Macros, and Host inventory. The 'Host' tab is active. Below the tabs, there are input fields for 'Host name' (containing 'ZBX Main') and 'Visible name' (containing 'Main Zabbix Server'). Under the 'Groups' section, 'Zabbix servers' is listed in the 'In groups' list. To the right, under 'Other groups', there are links for 'Discovered hosts', 'Linux servers', 'Templates', and 'Windows servers'. A green highlighted section labeled 'New host group' contains an empty input field. Below this, the 'Agent interfaces' section has a table with columns for 'IP address' (127.0.0.1) and 'DNS name' (zabbix.example.com), and a 'Connect to' column with 'IP' and 'DNS' buttons. There are 'Add' buttons for each interface type. Below the agent interfaces are sections for 'SNMP interfaces', 'JMX interfaces', and 'IPMI interfaces', each with an 'Add' button. At the bottom, there is a 'Monitored by proxy' dropdown menu set to '(no proxy)'.

In this case, `{HOST.HOST}` would resolve to ZBX Main, `{HOST.NAME}` to Main Zabbix Server, `{HOST.IP}` to 127.0.0.1, and `{HOST.DNS}` to zabbix.example.com. Finally, since the **Connect to** option is set to **IP**, `{HOST.CONN}` would resolve to 127.0.0.1 as well.

The most obvious use of these macros is to make trigger and graph names more dynamic and adaptable to the actual hosts they will be used into. Since a graph's name is displayed as a header when viewing the graph, it's vital to distinguish different graphs of the same type belonging to different hosts, especially when they are displayed together in a screen, as explained in *Chapter 5, Visualizing Data*.

A less obvious use of these macros is inside an item's key definition. We touched briefly on external scripts in *Chapter 4, Collecting Data*, and you'll meet them again in the next chapter, so we won't get into too much detail about them here. It's suffice to say that from an item creation point of view, all you need to know about an external script is its name and any parameters you may need to pass in order to execute it correctly.

Since external scripts, as is their nature, don't share any information with the rest of Zabbix other than the arguments they are passed and their return value, it's often essential to include the host's IP address or hostname as one of the arguments. This ensures that the script will connect to the right host and collect the right data. A single, well-configured script can perform the same operation on many different hosts thanks to the template systems and macros like `{HOST.CONN}`, `{HOST.IP}`, and so on.

Take, for example, a script that checks if a particular application is alive using a custom protocol. You could have an external script, let's call it `app_check.sh`, which takes a host name or IP address as an argument, connects to it using the application's protocol and returns 1 if it's alive and well, and 0 if the check fails. Your template item's key would look like the following screenshot:

The screenshot shows the 'Item' configuration form in Zabbix. The form is titled 'Item' and contains the following fields and sections:

- Host:** Template Application Server
- Name:** App check
- Type:** External check (dropdown)
- Key:** app_check.sh[{HOST.IP}] (with a 'Select' button)
- Type of information:** Numeric (unsigned) (dropdown)
- Data type:** Decimal (dropdown)
- Units:** (empty text field)
- Use custom multiplier:** ☐ (checked) with a value of 1
- Update interval (in sec):** 30
- Flexible intervals:** A table with columns 'Interval', 'Period', and 'Action'. The table is currently empty, showing 'No flexible intervals defined.'
- New flexible interval:** A section with fields for 'Interval (in sec)' (50), 'Period' (1-7,00:00-24:00), and an 'Add' button.
- Keep history (in days):** 90

In these cases, using a macro as the argument to the item's key is the only way to make an external check for a part of a template and is useful for any regular host.

Another example would be that of a bunch of Zabbix hosts that don't represent regular operating system machines, physical or virtual, but single applications or single database instances. In a scenario like this, all the application hosts would share the same connections and interfaces, those of the actual server hosting the applications, and they would be linked to a template holding only items relevant to application-level (or database-level) measurements. To keep things simple, let's say that you have an application server (Alpha) hosting three different applications:

- a document archival facility (doku)
- a customer survey form manager (polls)
- a custom, internal microblogging site (ublog)


For each of these applications you are interested, by and large, in taking the same measurements

- number of active sessions
- amount of memory consumed
- number of threads
- network I/O
- number of connections to the database


Again, for simplicity's sake, let's say that you have a bunch of external scripts that, given an IP address and an application name, can measure exactly the preceding metrics. External script keys tend to be easy to read and self-explanatory, but all of this can be equally applied to JMX console values, Windows performance counters, database queries, and any other kind of items.

One way to monitor this set up is to create only one host, Alpha, and, in addition to all the regular OS and hardware monitoring items, a number of items dedicated to application measurements, repeated for each one of them. This can certainly work, but if you have to add a new application, you'll need to create all the items, triggers, and graphs related to it, even if they differ from the rest by just the application's name.

Seeing as that is the only difference in an otherwise identical collection of entities, a more flexible solution would be to split the monitoring of every application to a different host and apply a common template.

 A host, from Zabbix's point of view, is just a collection of entities with one or more connection interfaces. It doesn't have to be an actual, physical (nor virtual!) machine with a regular operating system. Any abstract but coherent collection of metrics and a means to retrieve them can be configured as a host in Zabbix. Typical examples are applications, database instances, and so on.

Instead of creating many similar items, triggers, and so forth for the host Alpha, you would create a custom application template and fill it with items that would look like the following screenshot:

CONFIGURATION OF ITEMS							Create item
Items							
Displaying 1 to 5 of 5 found							
Filter							
« Template list Template: Template Application Server Applications (0) Items (5) Triggers (0)							
Graphs (0) Screens (0) Discovery rules (0)							
<input type="checkbox"/>	Wizard	Name 	Triggers	Key	Interval	History	
<input type="checkbox"/>		Amount of memory consumed		app_memory.sh[{HOST.IP}, {HOST.NAME}]	30	90	
<input type="checkbox"/>		Application I/O		app_io.sh[{HOST.IP}, {HOST.NAME}]	30	90	
<input type="checkbox"/>		Number of active sessions		active_sessions.sh[{HOST.IP}, {HOST.NAME}]	30	90	
<input type="checkbox"/>		Number of application threads		app_threads.sh[{HOST.IP}, {HOST.NAME}]	30	90	
<input type="checkbox"/>		Number of database connections		app_db_conn.sh[{HOST.IP}, {HOST.NAME}]	30	90	
Enable selected Go (0)							

You could then create one host for each application, with Alpha's IP address as the connection interface, and with the application's name as host name. Linking the template you just created to the hosts would give you the same basic results as before, but with much more flexibility; adding an application to be monitored now is a simple matter of creating a host and linking it to the correct template. If you move an application from one server to another, you just need to update its IP address. If you put all these application hosts in a separate group, you can even grant access to their monitoring data to a specific group of users, without necessarily giving them access to the application server's monitoring data. And, it goes without saying, adding, deleting, or modifying an entity in the template applies immediately to all the monitored applications.

User-defined macros

A special class of macros is user-defined, template- and host-level macros. You can configure them in the **Macros** tab of every host or template creation and administration form. They are quite simple, as they only provide a translation facility from a custom label to some predefined, fixed value:

Macro	Value	
{\$NODATA}	60	Remove
{\$PFREE}	5	Remove
{\$MACRO}	value	Remove

[Add](#)

[Save](#) [Clone](#) [Full clone](#) [Delete](#) [Delete and clear](#) [Cancel](#)

When used in a template, they prove quite useful in defining common thresholds for triggers, so if you need to modify a bunch of time-out triggers, you can just modify the `{ $NODATA }` macro instead of changing every single trigger that uses it. User-defined macros can be used everywhere that built-in macros can be used.

The usefulness is even greater when used in connection with nested templates, as we'll see in a short while.

Linking templates to hosts

To link a template to a host, you can either select the hosts you want to link from the template's configuration form, as we've seen in the *Creating templates* section, or you can choose the template you need for a host in that host's configuration form by going into the **Template** tab.

Once linked, a host will inherit all of the template's entities. *Previously existing entities with the same name will be overwritten*, but entities not included in the template will remain as they are and not touched in any way by the linking operation.

All entities will maintain their original template's name when displayed in the configuration section of the Web interface, even when viewed from a host configuration page. However, this doesn't mean that modifying them from a template's configuration tab is the same as modifying them from a linked host's configuration tab.

If you modify an entity (item, trigger, graph, and so on) from a template's configuration tab, the modifications will apply immediately to all linked hosts. On the other hand, if you modify a template entity from a particular host's configuration tab, the changes will only apply to that host, and not on a template level. While this can be useful to address any special circumstances for an otherwise regular host, it can also generate some confusion if you make many local changes that can become hard to keep track of. Moreover, not every aspect of a template entity can be modified at the host level. You can change the frequency of an item, for example, but not its key.

Unlinking a template from a host doesn't eliminate its entities, unless you unlink and clear it. Be careful with this operation as all the items' history and trends would become unavailable. If you have collected any actual data, it's probably better to just unlink a template from a host, and then disable any unused items and triggers, while retaining all of their historical data.

Nesting templates

Just as you can link a template to a host, you can also link a template to another template. The operation is identical to linking a template to a host; you navigate to the **Linked templates** tab in a template's configuration form, and choose the templates you want to link.

While this may seem an awkward operation, it can prove quite useful in two cases.

The first application of nested templates is to make user macros even more general. Since a template inherits all of its linked templates entities and properties, any custom macro will also be inherited and thus made available to the actual monitored hosts.

To make a concrete example, let's say you have a Template Macros template containing a `{ $PFREE }` user macro, with a value of 5. You could use this macro to represent the amount of free disk space, in percentages, to check against, or free available memory, or any other such threshold. This template could be linked to both Template OS Linux and Template OS Windows templates, and the `{ $PFREE }` macro used in these templates' triggers. From now on, if you ever need to modify the default value of the free disk space percentage to check against, you'll just need to change the original Template Macros template and the updated value will propagate through the linked templates down to the monitoring hosts.

A second, somewhat more limited but still useful, way to use nested templates is to extend the inheritance beyond macros, to all the other entities. This may become an advantage when you have a common set of features on a given technological layer, but different uses on other layers. Let's take for instance the case where you have a large number of virtually identical physical servers, that host just a couple of versions of operating systems (Linux and Windows, for simplicity's sake), but that in turn perform many different specialized functions, database, file server, web server, and so on.

You could certainly create a few monolithic templates with all the items you need for any given server, from hardware checks to OS checks to application specific ones. Or, you could create a sort of hierarchy of templates. A common, hardware level template that enables IPMI checks will be inherited by a couple of OS specific templates. These, in turn, will be inherited by some application specific templates that will have names, such as Linux Apache Template or Win Exchange Template. These templates will have all the items, triggers, and graphs specific to the applications they are meant to monitor, in addition to all the OS specific checks they have inherited from the OS level templates and the hardware specific ones they have inherited from the HW level templates. This means that, when creating a host, you will still just need to link it to a single template, but you'll also have a lot of flexibility in creating new templates and nesting them, or modifying existing ones in only one place and watching the changes propagate along the template-linking chain. This also means having maximum generality while still maintaining the ability to make host specific customizations if you need to.

Combining templates

Another way to make templates modular is to create specific templates for any given technological layer and product, but not link them in a hierarchy at the template level.

You can instead link them – as many as you want – directly to the host you need to monitor, as long as they don't have any conflicting or overlapping item names or keys. As in the preceding scenario, Host A could have linked an IPMI checks template, an OS Linux one, and an Apache server one, while Host B could also have an IPMI checks template and an OS Linux one, but then also have a PostgreSQL database template.

The end result is practically the same as the nested templates solution described previously, so which one should you choose? This is largely a matter of preference, but a possible criterion could be that if you have a relatively low number of low-level templates and good consistency in your hardware, OS, and technological configuration, the nested solution might be easier to manage. You'll only have to connect the templates once and then use them on a large number of hosts. This approach also works well with the host discovery facility as it keeps things simple when linking templates to hosts. If, on the other hand, you have a great number of low-level templates, and a great variability in your technological configuration and landscape, you may just as well pick and choose the templates you need when you create your hosts. Any preconfiguration, in fact, would only prove too rigid to be really useful. This approach works well if you want to always ponder how you are creating and configuring your hosts, and also need a great deal of local customization that would make any aggressive inheritance feature a moot point.

Discovering hosts

A third way to link templates to hosts is to let the server do it automatically by combining Zabbix's host-discovery facility with discovery actions.

Zabbix's discovery facilities consist of a set of rules that periodically scan the network, looking for new hosts, or disappearing ones, according to predetermined conditions.

The three methods in Zabbix that can be used to check for new or disappeared hosts, given an IP range, are:

- Availability of a Zabbix agent
- Availability of a SNMP agent

- Response to simple external checks (FTP, SSH, and so on)
- These checks can also be combined, as illustrated in the following example:

The screenshot shows the 'Discovery rule' configuration window in Zabbix. The 'Name' field is set to 'Mail servers'. The 'Discovery by proxy' dropdown is set to 'No proxy'. The 'IP range' field contains '192.168.1.1-255'. The 'Delay (in sec)' field is set to '3600'. Under the 'Checks' section, three checks are listed: 'Zabbix agent "system.uname"', 'ICMP ping', and 'SMTP'. Each check has a 'Remove' link next to it. There is also a 'New' link. Under the 'Device uniqueness criteria' section, two options are shown: 'IP address' (selected with a radio button) and 'Zabbix agent "system.uname"' (unselected). The 'Enabled' checkbox is checked. At the bottom, there are four buttons: 'Save', 'Clone', 'Delete', and 'Cancel'.

As you can see, when enabled, this rule will check every hour, in the IP range 192.168.1.1-255, for any server that:

- Responds to a **ICMP ping** probe
- Has a correctly configured Zabbix agent that will return a value when asked for the `system.uname` item
- An **SMTP** listening port, which is usually associated to a mail server.

As usual with all things Zabbix, a discovery rule will not do anything by itself, except generate a discovery event. It will then be the job of Zabbix's actions facility to detect the said event and decide if and how to act on it. Discovery event actions are virtually identical to trigger event actions. As you've already seen trigger event actions in *Chapter 6, Managing Alerts*, following are the only differences when it comes to discovery events.

First, action conditions are a bit different, as can be expected, as shown in this following screenshot:

The screenshot shows the 'CONFIGURATION OF ACTIONS' dialog box. The 'Conditions' tab is selected. The 'Type of calculation' is set to 'AND / OR'. The 'Conditions' list contains three items:

Label	Name	Action
(A)	Received value like "Linux"	Remove
(B)	Discovery status = "Up"	Remove
(C)	Service type = "Zabbix agent"	Remove

The 'New condition' dropdown is open, showing a list of options: Host IP, Service type, Service port, Discovery rule, Discovery check, Discovery object, Discovery status, Uptime/Downtime, Received value, and Proxy. The 'Host IP' option is selected. The 'New condition' field shows 'Host IP = 192.168.0.1-127,192.168.2.1'. There are 'Save' and 'Cancel' buttons at the bottom.

Instead of host names and trigger specifications, you can set conditions based on things, such as **Discovery status**, **Service type**, and **Uptime/Downtime**. The **Received value** condition is of particular interest, as it allows things like differentiating between operating systems, application versions, and any other information you could get from a Zabbix or an SNMP agent query.

This kind of information will be critical when it comes to configuring the action's operations:

CONFIGURATION OF ACTIONS

Action

Conditions

Operations

Action operations

Details

Add to host groups: Linux servers

Link to templates: Template OS Linux

Operation details

Operation type

Send to User groups

Send to Users

Send only to

Default message

Add Cancel

Send message

Remote command

Add host

Remove host

Add to host group

Remove from host group

Link to template

Unlink from template

Enable host

Disable host

Action

Action

Save

Clone

Delete

Cancel

Sending a message and executing a remote command are the exact equivalent of the same operations available to trigger event actions. On the other hand, if adding (or removing) a host is quite a self-explanatory action, when it comes to adding to a host group, or linking to a template, it becomes clear that a good set of actions with specific received value conditions and template linking operations can give a high level of automation to your Zabbix installation.

This high-level of automation is probably more useful in rapidly changing environments that still display a good level of predictability depending on the kind of hosts you can find, such as fast growing grids or clusters. In these kinds of environments you can have new hosts appearing on a daily basis, and maybe old hosts disappearing at almost the same rate, but the kind of host is more or less always the same. This is the ideal premise for a small set of well-configured discovery rules and actions, so you don't have to constantly and manually add or remove the same types of hosts. On the other hand, if your environment is quite stable, or you have very high host-type variability, you may want to look more closely at what and how many hosts you are monitoring as any error can be much more critical in such environments.

On the other hand, limiting discovery actions to sending messages about discovered hosts can prove quite useful in such chaotic environments, or where you don't directly control your systems' inventory and deployment. In such a case, getting simple alerts about new hosts, or disappearing ones, can help the monitoring team keep Zabbix updated despite any communication failure between IT departments, accidental or otherwise.

Low-level discovery

An even more useful and important feature of Zabbix templates is their ability to support special kind of items, called low-level discovery rules. Once applied to actual hosts, these rules will query the host for whatever kind of resources they are configured to look for, filesystems, network interfaces, SNMP OIDs, and more. For every resource found, the server will then dynamically create items, triggers, and graphs according to special entity prototypes connected to the discovery rules.

The great advantage of low-level discovery rules is that they take care of the more variable parts of a monitored host, such as the type and number of network interfaces, in a dynamic and general way. This means that, instead of manually creating specific items and triggers of every host's network interfaces or filesystems, or creating huge templates with any possible kind of item for a particular operating system, and keeping most of these items disabled, you can have a reasonable number of general templates that will adapt themselves to the specifics of any given host, by creating on the fly, any entity needed based on discovered resources and previously configured prototypes.

Out of the box, Zabbix supports three discovery rules:

- Network interfaces
- Filesystems types
- SNMP OIDs

As discovery rules are effectively special kinds of items, you can create your own rules, provided you understand their peculiarity compared to regular items.

If we don't consider the fact that you need to create and manage low-level discovery rules in the **Discovery rules** section of template configuration, and not in the usual **Items** section, the main difference between the two kinds of items is that while a regular item usually returns a single value, as explained in *Chapter 4, Collecting Data*, a discovery item always returns a list, expressed in JSON, of macro value pairs. This list represents all the resources found by the discovery items, together with means to reference them.

The following table shows Zabbix's supported discovery items and their return values, together with a generalization that should give you an idea on how to create your own rules:

Discovery item key	Item type	Return values
vfs.fs.discovery	Zabbix agent	<pre>{ "data": [{ "#FSNAME": "<path>", "#FSTYPE": "<fstype>" }, { "#FSNAME": "<path>", "#FSTYPE": "<fstype>" }, { "#FSNAME": "<path>", "#FSTYPE": "<fstype>" }, ...] }</pre>
net.if.discovery	Zabbix agent	<pre>{ "data": [{ "#IFNAME": "<name>" }, { "#IFNAME": "<name>" }, { "#IFNAME": "<name>" }, ...] }</pre>
snmp.discovery	SNMP (v1, v2 or v3) agent	<pre>{ "data": [{ "#SNMPINDEX": "<idx>", "#SNMPVALUE": "<value>" }, { "#SNMPINDEX": "<idx>", "#SNMPVALUE": "<value>" }, { "#SNMPINDEX": "<idx>", "#SNMPVALUE": "<value>" }, ...] }</pre>


Discovery item key	Item type	Return values
custom.discovery	any	<pre>{ "data": [{ "#CUSTOM1": "<value>", "#CUSTOM2": "<value>" }, { "#CUSTOM1": "<value>", "#CUSTOM2": "<value>" }, { "#CUSTOM1": "<value>", "#CUSTOM2": "<value>" }, ...] }</pre>



As with all SNMP items, an item key is not really important, as long as it is unique. It's the SNMP OID value that you ask an agent that makes the difference, you can create different SNMP discovery rules that look for different kind of resources, by changing the item key and looking for different OID values. The custom discovery example is even more abstract, as it will depend on the actual item type.

As you can see, a discovery item always returns a list of values, but the actual contents of the list change depending on what resources you are looking for. In the case of a filesystem, the returned list will contain values like `{#FSNAME}:/usr`, `{#FSTYPE}:btrfs`, and so on for every discovered filesystem. On the other hand, a network discovery rule will return a list of the names of the discovered network interfaces.

When configuring a template's discovery rules, you don't need to care about the actual values returned in such lists, nor the lists' length. The only thing you have to know is the name of the macros that you can reference in your prototypes. These are the second half of the low-level discovery mechanisms. You create them as regular template entities, making sure you use the discovery item macros where needed, as exemplified in the following screenshot:

CONFIGURATION OF ITEM PROTOTYPES						
Item prototypes of Mounted filesystem discovery						
Displaying 1 to 5 of 5 found						
« Template list Template: Template OS Linux « Discovery list Discovery: Mounted filesystem discovery						
Trigger prototypes (2) Graph prototypes (1)						
<input type="checkbox"/>	Name 	Key	Interval	History	Trends	Type
<input type="checkbox"/>	Free disk space on {#FSNAME}	vfs.fs.size[{#FSNAME},free]	60	7	365	Zabbix ag
<input type="checkbox"/>	Free disk space on {#FSNAME} (percentage)	vfs.fs.size[{#FSNAME},pfree]	60	7	365	Zabbix ag
<input type="checkbox"/>	Free inodes on {#FSNAME} (percentage)	vfs.fs.inode[{#FSNAME},pfree]	60	7	365	Zabbix ag
<input type="checkbox"/>	Total disk space on {#FSNAME}	vfs.fs.size[{#FSNAME},total]	3600	7	365	Zabbix ag
<input type="checkbox"/>	Used disk space on {#FSNAME}	vfs.fs.size[{#FSNAME},used]	60	7	365	Zabbix ag
Activate selected ▼ Go (0)						

When you apply the template to a host, it will create items, triggers, and graphs based on the resources discovered by the discovery items and configured according to the discovery prototypes.

Custom discovery rules, from this point of view, work exactly in the same way as custom items, whether you decide to use agent-side scripts (thereby using a custom `zabbix.agent` item key), external scripts, database queries, or anything else. The only things you have to make sure of is that your custom items' return values must respect the JSON syntax as shown in the preceding table, and that you reference your custom macros in the entities' prototypes that you will create.

Summary

This chapter concludes the central part of the book, dedicated to developing a deeper understanding of the Zabbix monitoring system's core functions. The effective configuration and use of templates builds on all the knowledge gained from using and analyzing items, graphs and screens, triggers, and actions. To this knowledge, this chapter has added a few template-specific aspects that should help to tie all of the previous chapters together. At this point of the book, you should be able to perform all the tasks associated with implementing and managing a monitoring system, from choosing what to monitor and how to configure different item types, to putting together information-rich visualization items. You should also be able to select the triggers and actions that are most significant in order to maximize the expressiveness of your alerts while avoiding false positives. Finally, you should not have any problems bringing it all together through the use of macros, nested, and combined templates, in order to apply a consistent and meaningful configuration to a wide range of hosts, and to further automate these operations through host-level discovery actions and the low-level discovery facilities of Zabbix's templates.

The final part of the book will be about further customization options for Zabbix, how to extend its functionality, and how to really integrate it with the rest of your IT management systems, in order to bring out its full power.

The next chapter will focus on writing extension scripts for Zabbix and its monitoring protocol.

8

Handling External Scripts

So far, you have learned how most part of a server's components work and how to leverage Zabbix to acquire data from various external sources. Considering that, set up your monitoring system in a large heterogeneous and complex infrastructure. Most probably, you will find a different custom device, server appliance, and proprietary hardware. Usually, all those devices have an interface to be enquired; but unfortunately, it often happens that most of the metrics are not exposed via **Simple Network Management Protocol (SNMP)** or any other standard query method.

Let us consider a practical example. Nowadays, all the UPSs own a temperature sensor and if you are in a complex infrastructure, it is possible that those UPSs are custom made, out of standard, and most probably, this sensor can be enquired only with a tool provided from the UPS vendor. Now the temperature of a UPS is a critical parameter, especially if the UPS is a big, custom made UPS. It is really important to monitor these metrics.

Imagine that your cooling system is not working properly; receiving an alarm just when the temperature reaches over the warning level is fundamental. On the other hand, predicting the failure will save a lot of money. Also, even if the physical damage is not really expensive, the downtime can cost a lot of money and have a terrible impact on your business. A good example is the case of a trading company. In this scenario, everything should be in perfect working order. In this environment, there is terrible competition to achieve better performance against competitors – buying a stock option some milliseconds before the others is a big advantage. Here it is easy to understand that if servers are not performing well, it is already an issue; if they are down, it is a complete disaster for the company. This example explains how critical it is to predict a failure.

Moreover, it is important to understand how critical it is to retrieve all functioning parameters of your infrastructure. This is where Zabbix comes to the rescue, providing interesting methods to retrieve data that interacts with the operating system, eventually enabling you to use a command-line tool. Zabbix's responses to this kind of requirement are as follows:

- External checks (server side)
- UserParameter (agent side)
- Zabbix_sender binary (can be used on both agent and server side)
- A simple, efficient, and easy-to-implement communication protocol

This chapter will entirely explore those alternatives to interact with the operating system and receive data from external sources. In this chapter, you will learn that there isn't a general, optimal, valid solution for all the problems, but each of them has its pros and cons. This chapter will make you aware of all the things that need to be considered when you implement a custom check. The analysis proposed will enable you to choose the best solution for your problems.

This chapter will cover the following points:

- Writing a script and making it available as external scripts
- Advantages and disadvantages of scripts on the server side or agent side
- Exploring some alternative methods to send data to your Zabbix server
- Detailed documentation about the Zabbix protocol
- Some commented educational implementation of the Zabbix sender protocol

External checks

Zabbix provides some features to cover all the items that cannot be retrieved with the standard agent. In real life, it is possible that you are not able to install the standard agent on the device that you would like to monitor. A practical example is the UPS or all the servers that for some reason cannot be compromised when installing external software, or all the appliances that cannot have custom software installed.

Now for all those reasons, you cannot have an agent on your device but you need to monitor some of the vital parameters of this devices, the only feasible solution is to use an external check.

The script's placement

The script's location on Zabbix is defined into the `zabbix_server.conf` configuration file. Since Zabbix Version 2.0, the default location has changed to `/usr/local/share/zabbix/externalscript`.



The default location depends on compile time from the variable `datadir`. Actually, the default location is `${datadir}/zabbix/externalscripts`. This rule is valid for both proxy and server components.

Previously, it was defined as `/etc/zabbix/externalscripts`; anyway, you can change it by simply specifying a different location on `zabbix_server.conf`, using the `ExternalScript` parameter.

```
### Option: ExternalScripts
# Mandatory: no
# Default:
# ExternalScripts=${datadir}/zabbix/externalscripts
ExternalScripts=/usr/local/zabbix/externalscripts
```



Zabbix Version 2.0 also accepts an empty string as a valid value. In the previous releases, an empty value was treated as an unsupported item causing issues.

There are some important enhancements to external script functionality introduced in Zabbix 2.0

- Key syntax now supports multiple comma-separated parameters
- Supports multilines values

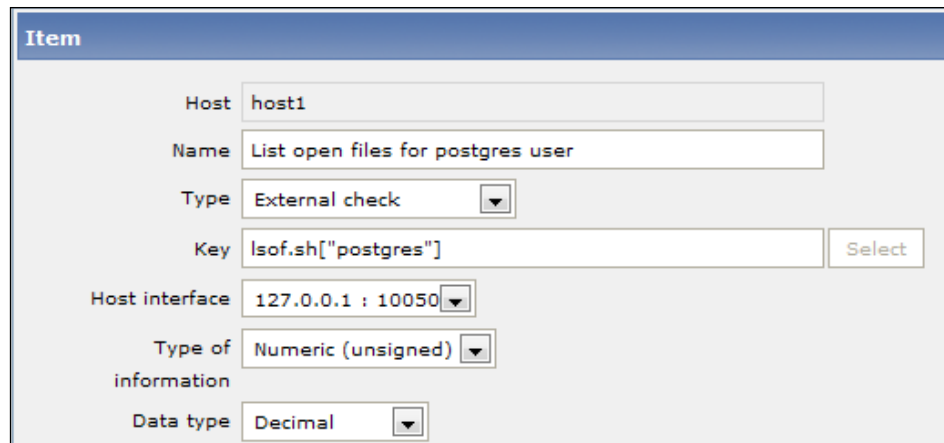
Going deep into external checks

Now it is time for a practical example. This is an easy example to understand, how External Scripts works. In the following example, we will count the number of open files for a specified user. The first thing to do is create the script and place it in the `ExternalScripts` location. The script will be called `lsuf.sh` and will contain the following code:

```
#!/bin/bash
if grep -q $1 /etc/passwd
then lsuf -u $1 | tail -n +2 | wc -l
else
echo 0
fi
```

This software requires the username as an input parameter, check if the username exists on the system and then return the number of open files for that account.

Now you only need to create a new item of type External check. In the **Key** field, enter `lsuf.sh["postgres"]` as shown on the following screenshot:



Item	
Host	host1
Name	List open files for postgres user
Type	External check
Key	lsuf.sh["postgres"] Select
Host interface	127.0.0.1 : 10050
Type of information	Numeric (unsigned)
Data type	Decimal

Now, on navigating to **Monitoring | Latest Data** it is possible to see the data retrieved by our script.

List open files for postgres user	03 Nov 2013 11:06:18	116	+86	Graph
-----------------------------------	----------------------	-----	-----	-----------------------

Now that you know how ExternalScript works, it is time to see how we can implement something more complex thanks to this feature.

In the next example, we will monitor some remote Oracle instances. There are some prerequisites to have this setup fully working: an Oracle client installed with a working `sqlplus` and `tnsping` and an account configured on your Oracle database target.

The last version of this software is available for download at http://www.smartmarmot.com/product/check_ora.

Anyway, it is interesting to see how it evolved from Version 1.0. Version 1.0 is available for download directly on the Zabbix forum available at <https://www.zabbix.com/forum/showthread.php?t=13666>.

This script is a good example of an external check. Basically, to have everything properly configured you need to do the following:

1. Create a user account on all your monitored databases.
2. Configure your Oracle client.
3. Decompress the package on your external script location.
4. Configure your database account on the following location:

`<EXTERNAL_SCRIPT_LOCATION>/check_ora/credentials`

5. Create a host with the same name as your database instance

The last point is of particular importance and it is a particular mode to use Zabbix. This method can be reused everytime that you need to aggregate some metrics that are not tied to a real host but to a service. To do a practical example, if you have a DBMS that can failover against another server, you can simply create a Zabbix fake host, called with the same name as that of your database. Now if the services do failover, you don't have an interruption on your collected data because the failover process is transparent from the server that is providing the service. This method is applied here because the Oracle client will handle a failover automatically, once properly configured.

Now, you can go ahead and create a host with the same name as that of your SID, for example, you have an Oracle instance to monitor that is defined as ORCL on your `tnsnames.ora`; thus, the Zabbix host will be ORCL.



You can create hosts tied to the name of the service; this enables you to abstract the service from the host that is providing the service.

The detailed configuration of an Oracle client is out of the scope of this book. Once you complete the configuration, you can test the script by simply running the following command:

```
check_ora.sh[-i <instance> -q <query>]
```

In the preceding command line, `<instance>` represents your instance name and `<query>` is the query file that you would like to run. There is a large amount of query files prebuilt in the `check_ora` directory, you can check all of them against your database.



The usage of Oracle SID or an Oracle instance name like the hostname on Zabbix is really useful here. It can be expanded by the {HOSTNAME} macros, so you can simply create a key such as `check_ora.sh [-i {HOSTNAME} -q query]` on your template and it will be expanded across all your databases.

Now in the Zabbix's host, you need to create the item to retrieve the external check and the key will be as follows:

```
check_ora.sh[-i {HOSTNAME} -q <queryhere>]
```

For example:

```
key="check_ora.sh[-i {HOSTNAME} -q lio_block_changes]"
```

The template is available on the forum at the same location. Note that {HOSTNAME} is expanded with the hostname, which in our case is exactly the Oracle instance name. You can have a generalized template using the {HOSTNAME} macro and their items are propagated across all your databases' hosts.

Now, the lifecycle of this item will be the following:

1. Zabbix calls the script.
2. The script will perform the following:
 - Login to the database
 - Execute the query and retrieve the value
 - Return the value on the standard output, Zabbix will receive the value and if it is valid, it will be stored

Inside the script

The core function of `check_ora.sh` is the `execquery()`. The function is the following:

```
execquery () {
start_time=$(date +%s)
#      echo "Time duration: $((finish_time - start_time)) secs."
echo "BEGIN check_ora.sh $1 $2 `date`" >> /tmp/checkora.log
  cd $SCRIPTDIR;
  sqlplus -S $1 <<EOF | sed -e 's/^\\ *//g'
set echo off;
set tab off;
set pagesize 0;
set feedback off;
```

```

set trimout on;
set heading off;
ALTER SESSION SET NLS_NUMERIC_CHARACTERS = '.,';
@$2
EOF
finish_time=$(date +%s)
echo "END check_ora.sh $1 $2 `date`" >> /tmp/checkora.log
echo "check_ora.sh $1 $2 Time duration: $((finish_time - start_time))
secs." >> /tmp/checkora.log
}

```

This function will begin producing some log information on /tmp/checkora.log

```

start_time=$(date +%s)
#      echo "Time duration: $((finish_time - start_time)) secs."
echo "BEGIN check_ora.sh $1 $2 `date`" >> /tmp/checkora.log

```

Those are useful to understand which external check is on-going and against which database. Plus in the logfile, you will find the elapsed time for the whole operation:

```

finish_time=$(date +%s)
echo "END check_ora.sh $1 $2 `date`" >> /tmp/checkora.log
echo "check_ora.sh $1 $2 Time duration: $((finish_time - start_time))
secs." >> /tmp/checkora.log
}

```

Since this file is shared (between the `check_ora.sh` **process**), and the Zabbix calls are not serialized, it is important to report the script calling line twice so that you can identify exactly which starting line correspond to which finish line. Here, to avoid any doubt, the elapsed is calculated and reported on the finish message.

After the script, call `sqlplus`

```
sqlplus -S $1 <<EOF | sed -e 's/^\s */g'
```

Here, `sed` cleans up the white space at the beginning of the output. This is because the returned data is a number that cannot begin with blank spaces; if that happens, the item will become unsupported!

The following code snippet makes an Oracle client less verbose:

```

set echo off;
set tab off;
set pagesize 0;
set feedback off;
set trimout on;
set heading off;

```

The preceding lines are important to avoid noise in the output.

The following code snippet explains the separator that should be used:

```
ALTER SESSION SET NLS_NUMERIC_CHARACTERS = '.,';
```

This section is important because you can have databases installed for different reasons with different character sets. Also, the client can use a different separator for decimals. You need to avoid all the uncontrolled charset conversions, and this is a general rule.

Finally, the script executes the query file in the following way:

```
@$2  
EOF
```

The output is returned in a standard output and is collected from Zabbix.

General script writing rules

This script covers all the critical points that you need to pay attention to:

- Do not introduce unwanted characters into the output
- Be aware of the type, so if a number is expected, remove all the unneeded characters (such as heading spaces)
- Avoid local conversions of numbers, the case of the dot and comma is a good example
- Have a log, keeping in mind that external script are not serialized, so you can have your log messages mixed in your logfile
- Those scripts of course run with the Zabbix server user, so maybe you need to take care of file permissions

Some consideration about external checks

In this section, you have seen how external checks can be executed and a complex task, such as database monitoring, is handled with them. If you have few external checks to implement, this can be a feasible solution to retrieve some metrics. This kind of approach with external checks, unfortunately, is not the solution to all the problems. On the other hand, you need to consider that they are really resource intensive and were once widely applied. Since external checks are on the server side, it is better not to overload the Zabbix server. The Zabbix server is the core component of your monitoring infrastructure and you can't steal resource from this server.

The UserParameter

The simple thing to do is to avoid extensive resource usage by your script by placing the script on the agent side. Zabbix provides this alternative method, and the script should instead be on server side and load the Zabbix server; it can be offloaded to the agent side with `UserParameters`.

`UserParameter` are defined on the agent configuration file. Once they are configured, they are treated like all the other Zabbix agent items by simply using the key specified in the parameter option. To define a user parameter, you need to add on the agent configuration file something like the following:

```
UserParameter=<key>,<shell command>
```

Where `key` must be unique and the `shell` command represents the command to execute. The command can be specified here inline and doesn't need to be a script like in the following example:

```
UserParameter=process.number, ps -e |wc -l
```

In this example, the `process.number` key will retrieve the total amount of the process number on your server.

With the same kind of approach, you can check the number of users currently connected with the following code:

```
UserParameter=process.number, who |wc -l
```

The flexible UserParameter

It is easy to understand that using this method you are going to define a huge amount of entry inside the agent configuration file. This is not the right approach because it is better keep the configuration file simple.

Zabbix provides an interesting `UserParameter` feature to avoid the proliferation of those items on the agent side, the flexible user parameter. This feature is enabled with an entry of this kind:

```
UserParameter=key[*],<shell command>
```

Here, the `key` still needs to be unique and the `[*]` term defines that this key accepts the parameters. The content between the square bracket is parsed and substituted with `$1...$9`; please note that the `$0` refers to the command itself.

An example of UserParameter can be the following:

```
UserParameter=oraping[*],tnsping $1 | tail -n1
```

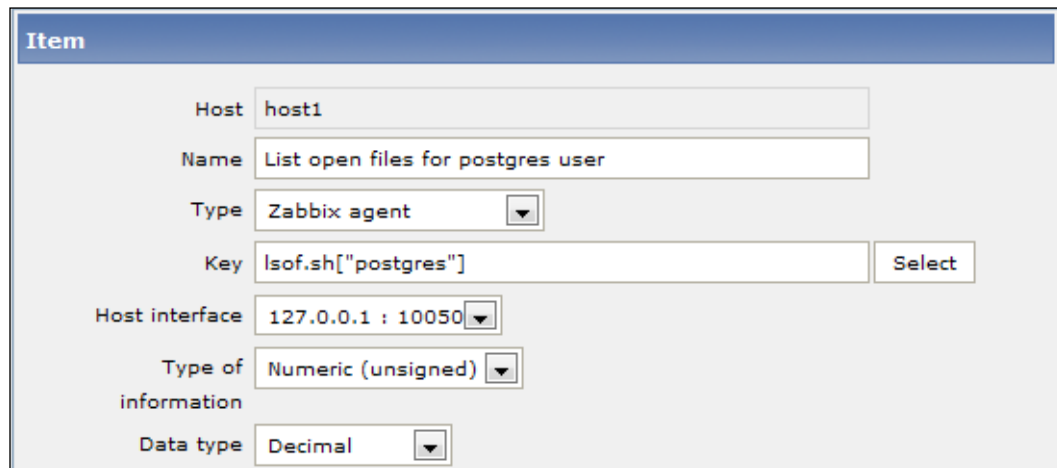
This command will execute the `tnsping` to your SID, passing it as `$1`. You can apply the same method in the the process for counting specified users as follows:

```
UserParameter=process.number[*], ps -e |grep ^$1 | wc -l
```

Then, if we want to move to the agent side, for the first script that returns the number of open files for a defined user, the configuration will be the following:

```
UserParameter=lsof.sh[*],/usr/local/bin/ /lsof.sh $1
```

Once added, you only need to restart the agent. On the server side, you need to switch the item **Type** to **Zabbix agent** and save it.



The screenshot shows the 'Item' configuration window in Zabbix. The fields are filled as follows: Host is 'host1', Name is 'List open files for postgres user', Type is 'Zabbix agent' (selected from a dropdown), Key is 'lsof.sh["postgres"]' with a 'Select' button to the right, Host interface is '127.0.0.1 : 10050' (selected from a dropdown), Type of information is 'Numeric (unsigned)' (selected from a dropdown), and Data type is 'Decimal' (selected from a dropdown).

With the same method, you can configure the `check_ora.sh` script to check the database with the following code:

```
UserParameter=check_ora.sk[*],check_ora.sh -i $1 -q $2
```

On the Zabbix server side, you need to create the item of Zabbix agent type or Zabbix agent (active) type and on the key you need to specify:

```
check_ora.sk[<dbname> <query_to_execute>]
```



You can test `UserParameter` using the command line, as previously described, or using the `zabbix_get` utility. With `zabbix_get`, you don't need to wait to see data between the latest data and it is easier to debug what is happening on the agent side.

There are methods to test if your `UserParameter` is working fine and the agent is able to recognize it. The first one is with `zabbix_get`, for example, in the case of `lsof.sh` from the Zabbix server, we can use the following:

```
# zabbix_get -s 127.0.0.1 -p 10050 -k lsof.sh["postgres"]
2116
```

The response is the result of the operation. Alternatively, we can log on to the monitored host and run the following command:

```
# /usr/sbin/zabbix_agentd -t lsof.sh["postgres"]
lsof.sh[postgres] [/usr/local/bin/lsof.sh postgres] [t|2201]
```

Again, this will display once the output and the script that is called.

Some considerations about UserParameters

With `UserParameter`, you moved the script from the server side to the agent side. The workload introduced by script is now on the agent side and you avoided resource stealing on the server side. Another point to consider is that this approach divides the workload between multiple servers. Obviously, each agent will monitor the database present on its hosts.

The `UserParameter` parameters are really flexible. To enable them on the agent side, you need to change the configuration file and restart the agent. Also, here you need to be sure that the returned value is properly set or it will be discarded.

Now, between the cons you need to consider the observer effect (discussed in *Chapter 1, Deploying Zabbix*) introduced with this kind of monitoring. You need to keep things as lightweight as possible, especially because the agent runs on the same server that provides the service.

UserParameter usage implies that you need to distribute the scripts and the relative updates across all your servers. In this example, where you want to monitor Oracle, you need to consider how many different versions of operating systems and software you need to handle. It is possible that in some time you will need to handle a myriad of different flavors of your scripts and software. This myriad of script, version, and so on will force you to have a centralized deployment, that is, all the versions of the scripts are stored in a centralized repository and so on. In addition, you need to take care of the workload added by your scripts and, if they don't handle all the possible exceptions well, this can be really complex to manage.

UserParameter is really good, flexible, and sometimes indispensable to solve some monitoring requirements, but are not designed for a massive monitoring against the same host. For all these reasons, it is time to explore another way to massively monitor the items that Zabbix doesn't support natively.

The following are some very important points about external scripts and Userparameters:

- All input is passed as parameters to the script and should properly be sanitized within the script to prevent command injection.
- All values are returned via `STDOUT` and should be in the format of the expected return type. Returning nothing will cause the Zabbix server to flag this item as unsupported.
- Make sure that all scripts terminate in a short amount of time.
- Make sure that scripts do not share or lock any resources, or have any other side effects, to prevent race conditions or incorrect interactions from multiple executions.

Sending data using zabbix_sender

Until here, you have seen how to implement external checks on both the server and agent side, moving the workload from the monitoring host to the monitored host. You can understand how both methods in the case of heavy and extensive monitoring are not the best approach, since we are thinking of placing Zabbix in a large environment. Most probably, it is better to have a server dedicated to all our checks and use those two functionalities for all the checks that are not widely run.

Zabbix provides utility designed to send data to the server. This utility is `zabbix_sender`, and with it, you can send item data to your server, using the items of a Zabbix trapper type.

To test the `zabbix_sender` utility, you simply add a Zabbix trapper item to an existing host and run the command:

```
zabbix_sender -z <zabbixserver> -s <yourhostname> -k <item_key> -o
<value>
```

You will get a response like the following:

```
Info from server: "Processed 1 Failed 0 Total 1 Seconds spent 0.0433214"
sent: 1; skipped: 0; total: 1
```

You see, the `zabbix_sender` utility is really easy to use. That said, now we can dedicate a server to all our resource intensive scripts.

The new script

Now, we can change the script that has been previously used as an external check and `UserParameter` in a new version that sends traps to your Zabbix server.

The core part of the software will be as follows:

```
CONNECTION=$( grep $HOST\; $CONNFILE | cut -d\; -f2) || exit 3;
RESULT=$( execquery $CONNECTION $QUERY.sql);
if [ -z "$RESULT" ]; then
    send $HOST $KEY "none"
    exit 0;
fi
send $HOST $QUERY "$RESULT"
exit 0;
```

This code executes the following steps:

1. Retrieve the connection string from a file.

```
CONNECTION=$( grep $HOST\; $CONNFILE | cut -d\; -f2) || exit 3;
```
2. Execute the query specified into the `$QUERY.sql` file.

```
RESULT=$( execquery $CONNECTION $QUERY.sql);
```
3. Check the result of the query and if it is not empty, send the value to Zabbix; otherwise, the value is replaced with "none".

```
if [ -z "$RESULT" ]; then
    send $HOST $KEY "none"
    exit 0;
fi
send $HOST $KEY "$RESULT"
```

In this code, there are two main functions in play: one is the `execquery()` function that basically is not changed and the other is the `send()` function. The `send()` function plays a key role in delivering data to the Zabbix server.

```
send () {
    MYHOST="$1"
    MYKEY="$2"
    MYMSG="$3"
    zabbix_sender -z $ZBX_SERVER -p $ZBX_PORT -s $MYHOST -k $MYKEY -o
"$MYMSG";
}
```

This function sends the values passed using a command line like the one already used to test the `zabbix_sender` utility. The value sent on the server side will have the corresponding Zabbix trapper item of kind trapper and Zabbix will receive and store your data.

Now to automate the whole check process, you need a wrapper that polls between all your configured Oracle instances, retrieve the data, and send it to Zabbix. The wrapper acquires the database list and the relative credential to log in from a configuration file and you need to call your `check_ora_sendtrap.sh` script recursively.

Writing a wrapper script for `check_ora_sendtrap`

Since this script will run from `crontab`, as the first thing, it will properly set up the environment source a configuration file:

```
source /etc/zabbix/externalscripts/check_ora/globalcfg
```

Then, go down to the script directory. Please note that the directory structure has not been changed for compatibility purposes:

```
cd /etc/zabbix/externalscripts
```

Then, it begins to execute all the queries against all the databases:

```
for host in $HOSTS; do
    for query in $QUERIES; do
        ./check_ora_sendtrap.sh -r -i $host -q ${query%.sql} &
        sleep 5
    done;
    ./check_ora_sendtrap.sh -r -i $host -t &
    sleep 5
    ./check_ora_sendtrap.sh -r -i $host -s &
done;
```

Note that this script for each database executes all the queries and retrieves the tnsping time and the connection time. There are two environment variables that are used to cycle between hosts and queries; they are populated with two functions:

```
HOSTS=$(gethosts)
QUERIES=$(getqueries)
```

The `gethost` functions retrieve the database name from the configuration file:

```
/etc/zabbix/externalscripts/check_ora/credentials
gethosts () {
    cd /etc/zabbix/externalscripts/check_ora
    cat credentials | grep -v '^#' | cut -d';' -f 1
}
```

The `getquery` function goes down into the directory tree, retrieving all the query files present.

```
getqueries () {
    cd /etc/zabbix/externalscripts/check_ora
    ls *.sql
}
```

Now you only need to schedule the wrapper script on `crontab`

```
* /5 * * * * /etc/zabbix/externalscripts/check_ora_cron.sh
```

And your Zabbix server will store and graph data.



All the software discussed here is available on SourceForge <https://sourceforge.net/projects/checkora> released on GPLv3 and on <http://www.smartmarmot.com/>.

Pros and cons of a dedicated script server

With this approach, we have a dedicated server that retrieves data. This means you do not overload the server that provides your service or the Zabbix server, and this is really a good point.

Unfortunately, this kind of approach lacks flexibility, and in this specific case, all the items are refreshed every five minutes. On the other hand, with the external check or `UserParameter`, the refresh rate can vary and be customized per item.

In this particular case, where a database server is involved, there is an observer effect introduced by our script. The query can be as lightweight as you want, but to retrieve an item, `sqlplus` will ask Oracle for a connection. This connection will be used only for a few seconds, the time needed to retrieve the item, after which the connection is closed. All this workflow basically lacks connection pooling. Using connection pooling, you can sensibly reduce the observer effect on your database.



Reducing the overhead with connection pooling is a general concept and it is not tied with a vendor-specific database. Databases in general will suffer if they are hammered with frequent requests of a new connection and a close connection.

Pooling the connection is always a good thing to do in general. To better understand the benefit of this methodology, you can consider a complex network with a path that crosses different firewalls and rules before arriving to a destination; this is the clear advantage of having a persistent connection. To have a pool of persistent connections kept valid with keep-alive packed, reduce the latency to retrieve the item from your database, and in general, the network workload. Creating a new connection involves the approval process of all the firewalls crossed. Also, you need to consider that if you are using Oracle, first is a connection request made against the listener that will require a call back once accepted and so on. Unfortunately, the connection pooling can't be implemented with the shell components. There are different implementations of connection pooling, but before we go deep into the programming side, it is time to see how the Zabbix protocol works.

Working with Zabbix protocols

The Zabbix protocol is quite simple; this is a strong point because it is simple to implement your own custom agent or software that sends data to Zabbix.

Zabbix supports the different versions of protocols. We can divide the protocols into three families:

- Zabbix get
- Zabbix sender
- Zabbix agent

The Zabbix get protocol

The Zabbix get protocol is really simple and easy to implement. Practically, you only need to send data to your Zabbix server at the port 10050.

This protocol is so simple that you can implement it with a shell script as well!

This is a textual protocol and is used to retrieve data from the agent directly. [root@zabbixserver]# telnet 127.0.0.1 10050

Trying 127.0.0.1...

Connected to 127.0.0.1.

Escape character is '^['.

agent.version

ZBXD2.0.6Connection closed by foreign host.

This example shows you how to retrieve the agent version simply with a telnet. Please note that the data is returned with a header that is ZBXD, followed by the data that represent the actual response 2.0.6.

This simple protocol is useful to retrieve data directly from the agent installed into our server and use it in a shell script.

This protocol is useful to identify the agent version without logging on to the server, or check all the instances of `UserParameter` defined against an agent.

The Zabbix sender protocol

The Zabbix sender is a JSON-based protocol. The message composition is the following:

```
<HEADER><DATA_LENGTH><DATA>
```

The `<HEADER>` section is of five bytes and it is in the form ZBXD\x01. Actually, only the first four bytes are the header, the next byte is used to specify the protocol version. Currently, only Version 1 is supported (0x01 HEX).

The `<DATA_LENGTH>` section is eight bytes in length and in hex format. So, for instance, 1 is formatted as 01/00/00/00/00/00/00/00, an eight bytes (or 64 bit) number in a hex format.

It is followed by <DATA>. This section is expressed in the JSON format.



Zabbix from Version 2.0.3 can receive *only* 128 MB of data to prevent the server from running out of memory. This limit has been added to protect the server from crashes caused by a large amount of data input.

To send the value, the JSON message needs to be in the following form:

```
<HEADER><DATALEN>{
  "request": "sender data",
  "data": [
    {
      "host": "Host name 1",
      "key": "item_key",
      "value": "XXX",
      "clock": unix_time_format
    },
    . . . .
    {
      "host": "Host name 2",
      "key": "item_key",
      "value": "YYY"
    }
  ],
  "clock": unix_time_format
}
```

In the previous example, as you can see, multiple items are queued on the same message if they come from different hosts or referred to as different items keys.



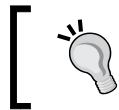
The "clock" term is optional in this protocol and can be omitted on the JSON object as well as at the end of the data section.

Once all the items are received, the server will send back the response. The response has the following structure:

```
<HEADER><DATALEN>{
  "response": "success",
  "info": "Processed 6 Failed 1 Total 7 Seconds spent 0.000283"
}
```

This example reports a response message; the following are some considerations:

- The response has a status that can be `[success|failure]` and it refers to the whole transmission of your items list to the Zabbix server.
- It is possible, like in this example, that some of the items failed. You simply receive a notification and you can't do much more than notify and write this status in a logfile.



It is important to keep track of the time spent to send your item list because if this value becomes high, or has a sensible variation, it means that our Zabbix server suffers on receiving items.

Unfortunately, this protocol does not give you feedback of which items failed and the reason for the failure. At the time of this writing, there were two requested features that are still pending:

- To have a more readable output, visit <https://support.zabbix.com/browse/ZBXNEXT-935>
- To identify the failed items, visit <https://support.zabbix.com/browse/ZBXNEXT-246>

Now you know how the Zabbix sender protocol works on Version 1.8 and later.

An interesting undocumented feature

There is an interesting sender's feature that is not widely known and not well documented. While going deep into the protocol analysis, the first thing to do is read the official documentation and the second is to check how Zabbix will implement it; because is it possible that not all the minor changes are updated in the documentation.

Then, looking into the `zabbix_sender` code, you can find the section where the protocol is implemented:

```
zbx_json_addobject(&senddval_args.json, NULL);
zbx_json_addstring(&senddval_args.json, ZBX_PROTO_TAG_HOST, hostname,
ZBX_JSON_TYPE_STRING);
zbx_json_addstring(&senddval_args.json, ZBX_PROTO_TAG_KEY, key, ZBX_
JSON_TYPE_STRING);
zbx_json_addstring(&senddval_args.json, ZBX_PROTO_TAG_VALUE, key_
value, ZBX_JSON_TYPE_STRING);
```

The preceding code snippet implements the Zabbix JSON protocol and in particular this section:


```
"host": "Host name 1",  
"key": "item_key",  
"value": "XXX",
```

Until here, the protocol has been well documented. Right after these lines are interesting sections that add one more property to our JSON item.

```
if (1 == WITH_TIMESTAMPS)  
    zbx_json_adduint64(&senddval_args.json, ZBX_PROTO_TAG_CLOCK,  
        atoi(clock));
```

Here a timestamp is provided within the item and is added as a property of the JSON object, after which the item is closed as follows:

```
zbx_json_close(&senddval_args.json);
```

[ The clock is defined as unsigned int64.]

This is a really important property because if you write your own `zabbix_sender`, you can specify the timestamp of when the item has been retrieved.

The important thing is that testing this section, Zabbix stores in its database the item, at the specified clock time.

Using clock properties in JSON items

Now this property can be used to optimize your sender. Zabbix supports 128 MB of data for a single connection. Of course, it is better to be far from that limit; because if we reach that limit, it is a sign that our implementation is not well done.

Clock can be used in two scenarios:

- If buffer items need to be sent and if they are sent inside a single connection in burst
- If the server is not available, you can cache and send the item later.

The first usage of this feature is clearly an optimization to keep the whole communication as lightweight as possible, reducing the number of connections against our Zabbix server can prevent issues.

The second way to enable this is implement a robust sender, which can handle a Zabbix server downtime and preserve your item in a cache, ready to be sent once the server is backed up and running. Please be aware to not flood the server if it is not reachable for a long period. Manage the communication by sending a reasonable number of items and not a long trail of items.

The Zabbix Agent protocol

This protocol is a bit more complex because it involves more phases and the dialogue is more articulated. When an active agent starts, the first thing it does is connect to the server and ask for a task to perform, in particular, which item should be retrieved and the relative timing.

Also, in the following, the form of the protocol is the same as used previously:

```
<HEADER><DATA_LENGTH><DATA>
```

Where <HEADER>, <DATA_LENGTH>, and <DATA> are as explained in the previous section.

The dialogue begins when the agent sends the following request to the server:

```
<HEADER><DATALEN>{
  "request": "active checks",
  "host": "<hostname>"
}
```

With this kind of request, the agent is going to ask in the active checklist for a specified hostname. The server response will, for instance, be as follows:

```
<HEADER><DATALEN>{
  "response": "success",
  "data": [{
    "key": "log[/var/log/localmessages,@errors]",
    "delay": 1,
    "lastlogsize": 12189,
    "mtime": 0
  },
  {
    "key": "agent.version",
    "delay": "900"
  }]
  "regexp": [
    {
      "name": "errors",
```

```
        "expression": "error",
        "expression_type": 0,
        "exp_delimiter": ",",
        "case_sensitive": 1
    }
}
```

Zabbix server must respond with success, followed by the list of items and the relative delay.

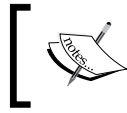


In the case of log and logrt items, the server should respond with lastlogsize. The agent needs to know this parameter to continue the work, also, mtime is needed for all the logrt items.

"regexp", which in this example is the response back to the agent, will exist only if you have defined some global or regular expression. Note that if a user macro is used, the parameter key is resolved and the original key is sent as key_orig. The original key is the user macro name.

Once the response is received, the agent will close the TCP connection and will parse it. Now the agent will start to collect the items at their specified period. Once collected, the items will be sent back to the server:

```
<HEADER><DATALEN>{
  "request": "agent data",
  "data": [
    {
      "host": "HOSTNAME",
      "key": "log[/var/log/localmessages]",
      "value": "Sep 16 18:26:44 linux-h5fr dhcpd[3732]: eth0: adding default
route via 192.168.1.1 metric 0",
      "lastlogsize": 4315,
      "clock": 1360314499,
      "ns": 699351525
    },
    {
      "host": "<hostname>",
      "key": "agent.version",
      "value": "2.0.1",
      "clock": 1252926015
    }
  ],
  "clock": 1252926016
}
```



While implementing this protocol, pay attention while sending back `lastlogsize` for all the log-type items and `mtime` for the `logrt` items.

The server will respond with

```
{
    "response": "success",
    "info": "Processed 2 Failed 0 Total 2 Seconds spent 0.000110"
}
```

Also, here there is a possibility that some items have not been accepted, but there isn't currently a way to know which ones they are.

Some more possible responses

To complete the protocol description, you need to know that there are some particular cases to handle:

- When a host is not monitored
- When a host does not exist
- The host is actively monitored but there aren't active items

In the first case, when a host is not monitored, the agent will receive the following response from the server:

```
<HEADER><DATALEN>{
    "response": "failed",
    "info": "host [Host name] not monitored"
}
```

In the second case, when the host does not exist, the agent will receive the following response:

```
<HEADER><DATALEN>{
    "response": "failed",
    "info": "host [Host name] not found"
}
```


And in the last case, when the host is monitored, but it does not have active items, the agent will receive an empty dataset:

```
<HEADER><DATALEN>{
    "response": "success",
    "data": []
}
```

Communicating with Zabbix

Now you know how the Zabbix protocol works and so it is time to see some code that implements this protocol. To keep things easy, we have described an example of the `zabbix_sender` protocol, the simplest way to send data to Zabbix.

Zabbix uses JSON to describe the object contained into data. There are a lot of efficient JSON libraries that can be used, but to make things more easy here, those libraries will not be used.

Implementing Zabbix sender in Java

Here you will see a really simple implementation of the `zabbix_sender` protocol that as you know, is the easy way to send a trap to Zabbix.

The piece of code that follows has been kept as simple as possible and the scope is to provide an example from which you can start to develop your own Zabbix monitoring component.

```
private String buildJsonString(String host, String item, Long
timestamp, String value){
    return    "{"
        + "\"request\": \"sender data\", \n"
        + "\"data\": [\n"
        +     "{\n"
        +         "\"host\": \"" + host + "\", \n"
        +         "\"key\": \"" + item + "\", \n"
        +         "\"value\": \"" + value.replace("\\", "\\") + "\", \n"
        +         "\"clock\": " + timestamp.toString()
        +     "}] \n" ;
}
```

This piece of code simply returns the JSON message to send it as a body. You only need to provide the host, item or better the item key, the value, and timestamp to include it into the message and it will return a JSON-formatted string object.

Now, once you have retrieved all your item values, you simply need to generate the JSON message, open a connection, and send the message. To open a connection against your Zabbix server, we can use the following lines of code:

```
String data = buildJsonString( host,item,value);
zabbix = new Socket(zabbixServer, zabbixPort);
zabbix.setSoTimeout(TIMEOUT);
out = new OutputStreamWriter(zabbix.getOutputStream());
int length = data.length;
```

In this code, as you see the program open a socket, define the timeout and retrieve the message length, now it is ready to send the message. Please remember that the message is composed with <HEADER><DATALEN><MESSAGE>, a simple way to send the header and the data length is the following:

```
out.write(new byte[] {
    'Z', 'B', 'X', 'D',
    '\1',
    (byte)(length & 0xFF),
    (byte)((length >> 8) & 0x00FF),
    (byte)((length >> 16) & 0x0000FF),
    (byte)((length >> 24) & 0x000000FF),
    '\0', '\0', '\0', '\0'});
```

This portion of code writes the message on the socket that actually contains the host, item, and value:

```
out.write(data);
```

Remember to flush the data and close the socket once the delivery is completed as follows:

```
out.flush();
out.close();
```

Now, we need to see what the Zabbix server will say about our item:

```
in = zabbix.getInputStream();
final int read = in.read(response);
String respStatus = (String) getValue(response);
if (read != 2 || respStatus.equals(ZBX_SUCCESS)) {
    in.close();
}
```

If the response is a success, you can close the InputStream.

This example is fully working but it is only for educational purpose. There are different things to improve before considering it ready for production. Anyway, this is a good starting point. This example can be extended by handling multiple JSON objects on the data section, increasing the number of objects passed per connection. You need to limit the connection numbers and avoid flooding your Zabbix server with connections just to send an item. Items can be buffered and sent together, for instance, if you have a group of items with the same schedule all of them can be sent together.

When you retrieve your items, it is important keep track of the timestamps. For doing so, you can add the timestamp to your item, and know when it has actually retrieved this metric.

In the previous example, the timestamp is not sent since it is optional, but it is a good practice to include it, especially if you're buffering some item; when you send it, the items will have the right timestamp.

Implementing Zabbix sender in Python

Nowadays, a lot of applications are written in Python and it is a programming language widely diffused and known. For this reason, it is an example of a fundamental treat that can be the starting point for your custom `zabbix_sender` in python. This piece of code can be extended and integrated directly into your software. Having a functionality integrated in the application can be really interesting because the application itself can send its health status to your Zabbix server. Now, it is time to take a look at the piece of code and how it works.

First, you need to define the structure and import `simplejson` used here, to add in JSON format the host, key, item value, and clock.

```
import simplejson as smplj
items_data = []
```

Now, retrieve the timestamp from the items; if it is null, we will get the current timestamp:

```
clock = zbxit.clock or time.time()
```

Now you can begin to create the JSON object to include it in the Zabbix message:

```
items_data.append(('\\t\\t{\\n'
                  '\\t\\t\\t"host":%s,\\n'
                  '\\t\\t\\t"key":%s,\\n'
                  '\\t\\t\\t"value":%s,\\n'
                  '\\t\\t\\t"clock":%s}') % (smplj.dump(zbxitem.
host), smplj.dump(zbxitem.key), smplj.dump(zbxitem.value), clock))
```

Now that your item has been transformed into a JSON object, it is time for the header:

```
json_items = ('{\\n'
              '\\t"request":"sender data",\\n'
              '\\t"data":[%s]\\n'
              '}') % ('\\n'.join(items_data))
```

The next step is to retrieve the length of our message to add it on the header:

```
data_len = struct.pack('<Q', len(json_items))
```

As previously discussed, here the message is put on the form <HEADER><DATALEN>+<JSON ITEM> as follows:

```
packet = 'ZBXD\\1' + data_len + json_items
```

Then, the socket is going to be open and the packet sent:

```
zabbix = socket.socket()
zabbix.connect((zabbix_host, zabbix_port))
zabbix.sendall(packet)
```

Once the packet has been sent, it is time to retrieve the Zabbix server response:

```
resp_hdr = _recv_all(zabbix, 13)
```

Next, check if it is valid:

```
if not resp_hdr.startswith('ZBXD\\1') or len(resp_hdr) != 13:
    return False
resp_body_size = struct.unpack('<Q', resp_hdr[5:])[0]
resp_body = zabbix.recv(resp_body_size)
zabbix.close()
resp = smplj.loads(resp_body)
if resp.get('response') != 'success':
    return False
return True
```

This piece of code is a good starting point to develop the Zabbix sender in python.

Some considerations about agent development

Now you probably don't see the time to begin the development of your software that sends a trap to Zabbix. But before beginning to write the code, it is fundamental to keep in mind the requirements and the problem.

Until here, you have two examples and you can easily start to send a trap to the Zabbix server, even if they are not completely engineered components.

First it is important to understand if it is only needed to send the data to Zabbix at a specified time schedule that is not directed from the Zabbix server. Those two pieces of code implement the Zabbix sender protocol, but the frequency with which the items are retrieved and sent can't be defined from the Zabbix server. Here, it is important to keep in mind who will drive your software, is it the Zabbix server or your software? To enable Zabbix to drive the sampling frequency, you need to implement the Zabbix agent protocol. The agent protocol is a bit more articulated and a bit more complex to implement. Anyway, the two examples proposed have all the components needed to properly handle the Agent protocol.

There is another point to consider. Usually, developers have their own preference for a programming language. Here, it is important to use the right instrument to solve the problem. A practical example would be to monitor your Oracle database. So your software will need to interact with the commercial software; the easy and logical choice is to use Java. Now, all the Python fans will stick up their nose! Here, more than the preference, it is important to keep in mind what is better supported from the monitored entity.

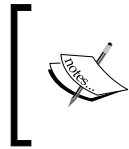
Oracle and databases in general produce standard industry engineered drivers for Java to interact with them. Most of the database vendors provide and, more importantly, update, fix, and develop their JDBC drivers continuously. It is better to delegate a bit of work to vendors. Also, they know their products better and you can get assistance on that.

Java has a lot of well-engineered components that will make your life easy in the difficult task of monitoring a database. For instance, the JDBC framework together with the database driver will provide an efficient connection pooling that can be configured to the following:

- Handle a minimum and maximum number of connections
- Validate the connection before using it for your software
- Send a keep-alive packet (useful to avoid firewall issues).

- Handle a reap time, removing all the idle connections (reducing the total amount of connection on the monitored server)

Those are only some of the points covered by JDBC all these points will help you to keep the monitoring lightweight and efficient.



An example of software made to monitor databases in general is DBforBIX available at <http://sourceforge.net/projects/dbforbix/> or <http://www.smartmarmot.com/product/dbforbix/>

Summary

In this chapter, we introduced you to all the possible ways that will help you interact with the server, enabling Zabbix to acquire items and metrics that are otherwise unsupported. In this chapter, we have seen the steps required to move the Oracle monitoring script from the server side to the client side, and then to its final destination, the dedicated server. Here, you have learned how a simple script grows until it becomes a complex external software. In each step, you have seen the progress and an analysis of the pros and cons of each location the script passed. This does not mean that you need a dedicated server for all your checks, but if your monitoring of the script is widely and extensively used, then it is a good practice. For each location passed, you have seen the positive side and the negative side of that particular placement. Now, you have a global vision of what can be done, and which one is the right place or point to act. Now the Zabbix protocol doesn't have more secrets and you can extend Zabbix ideally without any limits.

In the next chapter, you will learn how to interact with Zabbix using the API. The next chapter will explain how to take advantages of the Zabbix API for massive deployments of hosts, users, and massive and repetitive operations in general.

9

Extending Zabbix

Understanding the Zabbix monitoring protocol allows us to write scripts, agents, and custom probes. In other words, it allows us to freely extend Zabbix's monitoring capabilities by expanding its means to collect data.

When it comes to actually controlling and administrating its monitoring objects, though, the only point of access we have mentioned until now is the web frontend. Whether you need to add a user, change the sampling frequency of an item, or look at historical data, you always need to use the frontend as a user interface.

This is certainly a convenient solution for day-to-day activities, as all you need to have access to is a simple browser. The frontend itself is also quite powerful and flexible, as you can conveniently perform mass operations on many objects of the same type, and control different nodes and proxies from the same spot.

On the other hand, not every large and complex operation can be performed conveniently through a web application, and sometimes you don't need to just visualize some data, but you need to export it and feed it to other programs in order to further analyze it. This is where the Zabbix API comes in. As you will learn in this chapter, Zabbix's JSON RPC API provides all the functions available to the frontend, including user management, monitoring configurations, and access to historical data.

In the following pages, you will learn to:

- Write code to connect to the API and make queries through it
- Create custom operations to manage your installation
- Write complex and conditional mass operations
- Export monitoring data in a number of different formats

Let's start with a look at the general API architecture and the way to set up your code in order to interact with it.

Exploring the Zabbix API

Zabbix provides an entry point to interact, manipulate, configure, or create objects in Zabbix. This API is available through its PHP frontend at the following URL:

`http://<your-zabbix-server>/zabbix/api_jsonrpc.php`

The communication protocol is JSON based, and the medium used is obviously http/https.

Zabbix's JSON RPC API provides a nice interface and exposes a lot of functionality. Once authenticated, it will allow you to perform any kind of operation on the Zabbix objects. Now, if you need to configure Zabbix in a large or in a very large network, this Zabbix API can be really useful. As a practical example, you can consider that you may need to add a large amount of devices that most probably are already defined in a separate document. The API provides the entry point to add all of them in Zabbix by simply using a dedicated script.

Zabbix API was introduced with the Zabbix Version 1.8, and it went through some changes up until the current 2.0 Version. This version can be considered more stable and mature, but it is still officially in draft state, so things may change a little in the future versions. This does not mean that it's not suitable for a production environment; on the contrary, the bigger the installation, the more crucial can there be sensible use of the API to script complex and time-consuming operations.

The following code is a simplified JSON request to the Zabbix API:

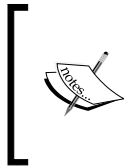
```
{
  "jsonrpc": "2.0",
  "method": "method.name",
  "params": {
    "param_1_name": "param_1_value",
    "param_2_name": "param_2_value"
  },
  "id": 1,
  "auth": "159121ba47d19a9b4b55124eab31f2b81"
}
```

The following points explain what the preceding code lines represent:

`"jsonrpc": "2.0"`: This is a standard JSON PRC parameter used to identify the protocol version; this will not change across your requests

"method": "method.name": This parameter defines the operation that should be performed, for instance, it can be `host.create` or `item.update`

- "params": This specifies the parameter needed by the method in JSON. Here if you want to create an item, the most common parameters, will be "name" and "key_"
- "id": 1: This field is useful to tie a JSON request to its response. Every response will have the same "id" provided in the request. This "id" is useful when you are going to send multiple requests at once if those requests don't need to be serialized or be sequential.
- "auth": "159121ba47d19a9b4b55124eab31f2b81": This is the authentication token used to identify an authenticated user; for more details please see the next section.



For a detailed description of all the possible parameters and methods, please refer to the official Zabbix documentation available at the following website:

<https://www.zabbix.com/documentation/2.0/manual/appendix/api/api>

Now it is important to remember that the whole communication usually is on HTTP. This is something to consider if we interact with Zabbix from our workstation or from a different network location. To interact with the Zabbix API, the first thing you need is authentication by the server, and here it is clear how important it is to have the whole communication encrypted and to use a secured channel. There are two different exposures for you to consider:

- Use http instead of https; otherwise the whole authentication will be in clear format, and readable
- Be aware of the sensitivity of the data being transmitted.

Now it is time to do the first step here with the API. The first thing you can do is the authentication.

Authenticating through the API

Here we discuss an example using a Unix shell because this will show us how simple communication is; later we will analyze an example with Python since it is widely used for rapid application development.

To test the authentication from our shell we can use `curl`. This is a common tool used to transfer data from/to a service using different protocols, and here we use `https` to make a secure channel. Now, for this test, you can log on to your Zabbix server and write the following command:

```
$ curl --insecure --include --netrc --request POST --header "Content-Type: application/json" --data@- https://127.0.0.1/zabbix/api_jsonrpc.php
```

Please note that `--insecure` specifies to `curl` to not verify the server certificate. This option produces a less-secure connection, but since we are the localhost, it is acceptable and will avoid a lot of certificate issues. Doing a practical example without the `--insecure` `curl` will respond with the following error:

```
curl: (60) Peer certificate cannot be authenticated with known CA certificates
```

More details here: <http://curl.haxx.se/docs/sslcerts.html>

Between the options, we set the Content-Type header as JSON and enable `curl` to receive data from the standard input with `--data@-`. Once this command is run you can paste the following JSON envelope:

```
{
  "jsonrpc": "2.0",
  "method": "user.login",
  "params": {
    "user": "Admin",
    "password": "my secret password"
  },
  "auth": null,
  "id": 0
}
```

Make sure you replace the `"password"` properties with your own password, and then you can close the standard input using `Ctrl + D`.


`curl` will take care to manage the whole https connection and will return the server's full HTTP response. In this case, we are interested in the authentication token:

```
HTTP/1.1 200 OK
```

The remaining output is the following:

```
Content-Type: application/json
{"jsonrpc": "2.0", "result": "403bbcdc3c01d4d6e66f68f5f3057c3a", "id": 0}
```

This response contains the token that we need to use for all the following queries on the Zabbix Server.

 The token will expire according to the auto-logout option set for the user who is authenticating.

Now to see how all this works, we can use `curl` again:

```
# curl --insecure --include --netrc -request POST --header "Content-Type:
application/json" --data@- https://127.0.0.1/zabbix/api_jsonrpc.php.
```

In this example, we are going to ask our server which is the last history value for the item "Processor load (15 min average per core)" then in this particular case, on this server, the JSON envelope will be the following lines of code:

```
{ "jsonrpc": "2.0",
  "method": "history.get",
  "params": {
    "output": "extend",
    "history": 0,
    "hostids": "10096",
    "itemid": "23966",
    "sortfield": "clock",
    "sortorder": "DESC",
    "limit": 1
  },
  "auth": "403bbcdc3c01d4d6e66f68f5f3057c3a",
  "id": 1
}
```

Remember that the request must contain the authentication token previously obtained using the "user.authenticate" method.

The server response in this case is the following:

```
HTTP/1.1 200 OK
```

...response header removed...

```
{ "jsonrpc": "2.0",  
  "result": [  
    { "hosts":  
      [ { "hostid": "10096" } ],  
      "itemid": "23840",  
      "clock": "1381263380",  
      "value": "0.1506",  
      "ns": "451462502" }  
    ], "id": 1 }
```

In this example, you have seen a way to use the authentication token to query the historical data for a particular host/item. Of course, shell scripting is not the best method to interact with Zabbix API because it requires a lot of coding to manage the "auth" token, and it is better to use something more user friendly.

Using the PyZabbix library

Now that we have a clear understanding of the API's architecture and its JSON-RPC protocol, we can move beyond the manual construction of the JSON objects and rely on a dedicated library. This will allow us to focus on the actual features of the API and not on the specifics of the implementation.


There are quite a few Zabbix API libraries available for different languages, but the one we'll use for the rest of the chapter is PyZabbix, written by *Luke Cyca* (<https://github.com/lukecyca/pyzabbix/wiki>). It's a small, compact Python module that stays quite close to the API while still being easy to use. Moreover, Python's interactive console makes it quite convenient to try some features and build a prototype before moving seamlessly to a complete script or application.

You can install PyZabbix very easily through Pip, the python package installer:

```
$ pip install pyzabbix
```

Once the module has been installed, you'll be able to import it and use it in your scripts to manage a Zabbix installation.

The first thing to do is to create an object for the API server and get an authentication token.

 The following code fragments are shown as part of an interactive session, but they can also be part of any Python code:

```
>>> from pyzabbix import ZabbixAPI
>>> zh = ZabbixAPI("https://127.0.0.1/zabbix/")
>>> zh.login("Admin", "zabbix")
```

Needless to say, you have to use your actual Zabbix frontend URL and user credentials for this code to work in your environment. If all goes well, this is actually all there is to it. From now on, you can use the object handler to access any API method in the following way:

```
>>> zh.host.get(output="refer")
```

The "refer" options will give you only the primary key and foreign key for any returned object:

```
[{'hostid': '9909900000010084'}, {'hostid': '9909900000010085'},
 {'hostid': '9909900000010086'}, {'hostid': '9909900000010087'},
 {'hostid': '9909900000010088'}]
```

Another advantage of using a Python library is that JSON datatypes map very cleanly onto Python ones, so much so that most of the time you won't even need to perform any additional type conversion. Here is a table that shows the specific types supported by the Zabbix API and some examples of how they look both in JSON and within PyZabbix function calls:

Type	JSON	pyzabbix
bool	<pre>{ "jsonrpc": "2.0" "method": "host.get", "params" : { "editable" : "true" } "auth" : <....> "id" : 1 }</pre>	<pre>zh.host. get(editable="true")</pre>

Type	JSON	pyzabbix
flag	<pre>{ "jsonrpc" : "2.0" "method": "host.get", "params" : { "countOutput" : "1" } "auth" : <....> "id" : 1 }</pre>	<pre>zh.host. get(countOutput=1)</pre>
integer	<pre>{ "jsonrpc" : "2.0" "method": "host.get", "params" : { "limit" : 10 } "auth" : <....> "id" : 1 }</pre>	<pre>zh.host.get(limit=10)</pre>
string	<pre>{ "jsonrpc" : "2.0" "method": "host.get", "params" : { "sortfield": "name" } "auth" : <....> "id" : 1 }</pre>	<pre>zh.host. get(sortfield="name")</pre>
timestamp	<pre>{ "jsonrpc": "2.0", "method": "event.get", "params": { "time_from": "1349797228", "time_till": "1350661228", }, "auth": <....>, "id": 1 }</pre>	<pre>zh.event.get(time_ from="1349797228", time_till= "1350661228")</pre>
array	<pre>{ "jsonrpc" : "2.0" "method": "host.get", "params" : { "hostids" : [1001, 1002, 1003] } "auth" : <....> "id" : 1 }</pre>	<pre>zh.host. get(hostids=[1001, 1002, 1003])</pre>

Type	JSON	pyzabbix
object	<pre>{ "jsonrpc" : "2.0" "method": "host.get", "params" : { "filter": { "name": ["Alpha", "Beta"] } "auth" : <....> "id" : 1 } }</pre>	<pre>zh.host. get(filter={"name": ["Alpha", "Beta"]})</pre>
query	<pre>{ "jsonrpc" : "2.0" "method": "host.get", "params" : { "output": "extend" } "auth" : <....> "id" : 1 } }</pre>	<pre>zh.host. get(output="extend")</pre>

The library creates the method requests on the fly, so it's fairly future-proof, which means any new or updated methods in the API will be automatically supported.

We can now move on to explore a few concrete examples of API usage. In order to keep the code more readable and to focus on the API, and not on general programming issues, all the examples will have a very simplistic and direct approach to data handling, without much data validation or error management. While you can certainly use the following fragments in interactive sessions or as part of more complex applications (or even to build a suite of dedicated command-line tools), you are strongly encouraged to make them more robust with the appropriate error handling and data validation controls.

Synchronizing templates across different nodes

One of the problems with a distributed node hierarchy that we talked about in *Chapter 2, Distributed Monitoring*, is that there is no built-in way to keep all templates synchronized across all nodes. You may recall that every node is an independent entity when it comes to configuration and object monitoring. Yes, it's possible to update a slave node's configuration from a master node, but it's not possible to transfer an existing configuration from a master node to a slave node. To do this, you need to manually replicate every change if you want to keep things consistent across nodes.

A possible solution is to use the export feature on your main node and then import and update your templates on all other nodes. In order to do that, you need to establish a workflow similar to this one:

- Maintain an updated list of nodes, along with API URLs and user credentials
- Update or create new templates only on the root node
- From the root node, export all templates
- Import exported templates in all nodes, updating existing ones and creating new ones

The two main limitations of this setup are that local template modifications on the different nodes will be overwritten, and as a consequence, only the users who have access to the root node will be able to define and distribute templates across the node hierarchy. Unless you have very specific needs, however, this should not really be a problem, and the distinct advantage of this setup is that it becomes very easy to automate the export-import process using the API. Let's see a simple script that does just that:

```
#!/usr/bin/python

from pyzabbix import ZabbixAPI

rootmasterUser = "Admin" masterPwd = "zabbix"
masterUrl = "https://127.0.0.1/zabbix"

nodefile = "nodes.csv"
with open(nodefile, 'r') as f:
    nodes = f.readlines()
```

The first few lines are used to set up a few variables, namely, the root node's connection information and the location of the URLs of other nodes. Be sure to put your actual URL and credentials instead of those shown here. In the `nodes.csv` file, you will put a line per Zabbix node, with a semicolon-separated list of node name, frontend URL, user, and password. Just remember to keep it updated and secured! The file is then read into a "nodes" array that will be used later to connect to the single nodes.

```
importRules = {"templates": {"createMissing": 'true',
                             "updateExisting": 'true'},
               "templateLinkage": {"createMissing": 'true'},
               "templateScreens": {"createMissing": 'true',
                                   "updateExisting": 'true'}}
}
```

`dict` will be used to define the entities the import method should actually import. If, for any reason, you just want to propagate new templates but don't want to lose any local modifications, just omit the `"updateExisting"` options.

```
masterZh = ZabbixAPI(masterUrl)
masterZh.login(user=masterUser, password=masterPwd)
```

As previously seen, these two lines connect to the main Zabbix server's API.

```
templates = masterZh.template.get(output=['templateid'])
tlist = []
for t in templates:
    tlist.append(t['templateid'])

exports = masterZh.configuration.export(format="json",
options={"templates": tlist})
```

Here we first get a clean list of template IDs and then use it to create an export of all the templates available.

```
for node in nodes:
    (name, url, usr, pwd) = node.rstrip('\n').split(";")
    nzh = ZabbixAPI(rootUrl)
    nzh.login(user=rootUser, password=rootPwd)
    result = nzh.confimport(format="json", rules=importRules,
source=exports)
    nzh.user.logout()
```

This is the main part of the script. For every node, we get its connection information, and then we connect to its API and execute a `configuration.import` method with the preceding options and the exported templates from the root node. Run this script every time you update a template or create a new one on the root node.

Needless to say, this is just the simplest possible version of such a script. Besides making it more robust with some error checking, you can add logging information or even periodically save every export in a file and only run the import part if two consecutive exports differ.

Mass operations

Another common use of the API facility is to automate some operations that you can perform from the web frontend, but they may be cumbersome or prone to errors. Things such as adding many users or updating the host IP addresses after merging two different networks fall under this category. The following fragments will assume that you already have a Zabbix API handle just as shown in the previous paragraphs. In other words, from now on, it will be assumed that your code will start with something like the following (remember that the Zabbix URL and user credentials here are just examples! Use your own URL and credentials):

```
#!/usr/bin/python
from pyzabbix import ZabbixAPI
user='Admin'
pwd='password'
url = 'https://127.0.0.1/zabbix/'
zh = ZabbixAPI(url)
zh.login(user=user, password=pwd)
```

Redistributing hosts to proxies

We have seen in *Chapter 2, Distributed Monitoring*, that you can add hosts to a proxy through the proxy configuration form or by updating every single host's `monitored by` property. Both of these methods can be too slow and cumbersome if you have a great number of hosts, and you need to update more than just a handful of them. If you just need to move an entire group of hosts from one proxy to another, you could also use the mass update functionality of the frontend, but if you need to distribute hosts to different proxies or work on just some hosts from many different groups, this approach won't scale well.

Here is one way to redistribute all the hosts monitored by a proxy to all the other proxies in your Zabbix installation. A possible reason to do this is that you may be doing some proxy maintenance, and you need to bring it down for a while, but you don't want to suspend monitoring for a whole bunch of hosts, so you redistribute them to other proxies.

First, let's get the proxy ID, knowing the proxy name:

```
proxy_name = "ZBX Proxy 1"
proxy_id = zh.proxy.get(filter={"host": proxy_name}, output="refer")
[0]['proxyid']
```

Once you have the proxy's ID, get the list of monitored hosts:

```
hlist = zh.proxy.get(selectHosts=['hostid'], proxyids=proxy_id,
output="refer")[0]['hosts']
hosts = [x['hostid'] for x in hlist]
```

Next, for simplicity's sake, let's just get the list of all other proxies excluding the one you are removing hosts from:

```
proxies = [x['proxyid'] for x in zh.proxy.get() if x['proxyid'] !=
proxyid]
```

Now we need to split the host list in as many roughly equal-sized chunks as the number of proxies available:

```
nparts = int(round(len(hosts)/len(proxies)))
hostchunks = [list(hosts[i:i+nparts]) for i in
range(0, len(hosts), nparts)]
```

The preceding code will divide your host list in as many sublists as the number of proxies you have. All that is left to do is to actually assign the hosts to the proxies:

```
for c in len(hostchunks):
    zh.proxy.update(proxyid=proxies[c], hosts=hostchunks[c])
```

And that's it. The `proxy.update` method will automatically reassign hosts, so you don't even have to remove them first from the original one. You can of course make things more robust by selecting only proxies in the same network as the one you are doing maintenance on, or by saving the host list so you can reassign it to the original proxy once it's available.

Adding or updating users

Even if you rely on some external authentication method for your Zabbix users, such as an LDAP server or Active Directory, no new user account will have any media information set nor will it belong to any group. This means that you'll still need to manually configure every user unless you create new users or update existing ones through some kind of code. For simplicity's sake, let's assume that you already have a list of usernames and email addresses, and the groups they should belong to, all gathered in a semicolon-separated `users.csv` file that looks like this:

```
adallevacche,a.dallevacche@example.com,Admins
slee,s.lee@example.com,Admins
jdoe,jdoe@foo.bar,DB admins; App Servers
mbrown,mbrown@example.org,Net admins
```

The script you are going to write will assume that the first field of every line will contain the username (called `alias` in the API). The second field will contain an e-mail address while the last field will be a comma-separated list of user groups that the user should belong to. Updating your users' information is quite simple. First, you need to read into your script the contents of the `users.csv` file:

```
with open('users.csv', 'r') as f:
    users = f.readlines()
```

Assuming that your Zabbix API connection object is called `zh`, you can now define a couple of helper functions and variables. The `mediatypeid` will be needed to update your users' media information. Assuming that you have only one e-mail media type defined in your Zabbix installation, you can get its ID by calling the following:

```
mediatypeid = zh.mediatype.get(output="refer",
    filter={"description": ['Email']})[0]['mediatypeid']
```

Unless you want to extend your `.csv` file with information about severity and the time period for each one of your users' media, you can also define a common template for all e-mail contacts:

```
def mkusermedia(mediatype='', email='', mediaid=''):
    return { 2mediaid2"mediaid": mediaid
            "mediatypeid": mediatype,
            "sendto": email,
            "active": 0,
            "severity": 63,
            "period": "1-7,00:00-24:00"
            }
```

Please note how 0 means *enabled*, while 1 means *disabled* for the active property. Also, while the `period` property is fairly self-explanatory, the `severity` property could look quite puzzling at first. It's actually a binary bitmap value and can be more easily understood if you take into consideration the trigger severity values and put them in order. Each severity level occupies a position of a six bits value

Severity	Disaster	High	Average	Warning	Information	Not classified
Enabled?	1	1	1	1	1	1
Decimal value	111111 = 63					

Since 63 equals 111111 in binary form, this means that the user will receive notifications for every severity level. If you want to receive notifications only for the disaster severity, you will have a 100000 bitmap and so a severity value of 32:

Severity	Disaster	High	Average	Warning	Information	Not classified
Enabled?	1	0	0	0	0	0
Decimal value	100000 = 32					

Similarly, to get notifications for disaster and higher levels, you'll need a 110000 bitmap and a severity value of 48, and so on.

Severity	Disaster	High	Average	Warning	Information	Not classified
Enabled?	1	1	0	0	0	0
Decimal value	110000 = 48					

The following helper function will get a list of group names and return a list of user group IDs that actually exist, ignoring non-existing group names:

```
def getgroupids(grouplist):
    return zh.usergroup.get(output=['usergrpid'], filter={"name":
grouplist.split(",")})
```

We can now proceed to actually work the user list to either update existing users or create new ones:

```
for u in users:
    (alias, email, groups) = u.split(",")
    user = zh.user.get(output=['userid'], filter={"alias": [alias]})
    if not user:
        zh.user.create(alias=alias,
                        passwd="12345",
                        user_medias=[mkus
usrgrps=getgroupids(groups),
ermedia(mediatype=mediatypeid, email=email)])
```

The `if` statement checks if the user exists. If not, the `user.create` method will take care of creating it, adding it to the appropriate groups, and creating the media contact as well. You need to define a password even if your users will authenticate from an external source. Depending on your password management policies, your users should be strongly encouraged to change it as soon as possible, or better yet, you could directly generate a random password instead of using a fixed string.

The second part of the `if` construct will get the `userid` and update the user's information.

```
else:
    userid=user[0]['userid']
    zh.user.update(userid=userid,
                   usrgprs=getgroupids(groups))
    usermedia = zh.usermedia.get(filter={"userid" : userid},
                                output=['mediaid'])
    zh.user.updatemedia(users = [userid],
                       medias=[mkusermedia(

mediaid=usermedia[0]['mediaid'],

mediatype=mediatypeid,

                                email=email))
```

Please note the way you need to call three different methods here for user groups and media, instead of just one. The first one will update group information; the second one will check for an already-defined e-mail address, and the third will update the said address or create a new one if it doesn't exist.

You can run the preceding code periodically to keep your user accounts updated. Obvious improvements would be adding each user's name and surname or getting user data directly from an LDAP server or any other source instead of from a `.csv` file.

Exporting data

Besides directly manipulating and monitoring internal objects, another compelling use of the Zabbix API is to extract data for further analysis outside of the Zabbix frontend. Maps, screens, graphs, triggers, and history tables can be excellent reporting tools, but they are all meant to be used inside the frontend. Sometimes you may need the raw data in order to perform custom calculations on it – especially when it comes to capacity planning – or you may need to produce a document with a few custom graphs and other data. If you find yourself with such needs on a regular basis, it makes sense to write some code and extract your data through the API. An interesting feature of the `get` methods, which are the fundamental building blocks of any data extraction code, is that they come with quite a few filters and options out of the box. If you are willing to spend some time studying them, you'll find that you'll be able to keep your code small and clean as you won't usually have to get lots of data to filter through, but you'll be able to build queries that can be quite focused and precise.

Let's see a few short examples in the following paragraphs.

Extracting tabular data

Zabbix provides a way to group similar items in a host in order to navigate them more easily when looking at the latest monitoring data. These item containers, called Applications, come in very handy when the number of items in a host is quite consistent. If you group all CPU monitoring items together under the label, say CPU, all filesystem-related items under Filesystems, and so on, you could find the data you are looking for more easily. Applications are just labels tied to a specific template or host and are used just used to categorize items. This makes them simple and lightweight. But it also means that they are not really used elsewhere in the Zabbix system.

Still, it's sometimes useful to look at the trigger status or event history, not just by host but by application too. A report of all network-related problems regardless of the host, host group, or specific trigger, could be very useful for some groups in your IT department. The same goes for a report of filesystem events, database problems, and so on.

Let's see how to build a script that will export in a `.csv` file all events related to a specific application. The setup is basically the same as the previous examples:

```
#!/usr/bin/python
from pyzabbix import ZabbixAPI
import sys
import csv
from datetime import datetime
appname = sys.argv[1]
timeformat="%d/%m/%y %H:%M"
zh = ZabbixAPI("http://localhost/zabbix")
zh.login(user="Admin", password="zabbix")
```

As you can see, the application name is taken from the command line while the API's URL and credentials are just examples. When you use your own you can also consider using an external configuration file for more flexibility. Since events are recorded using a Unix timestamp, you'll need to convert it to a readable string later on. The `timeformat` variable will let you define your preferred format. Speaking of formats, the `.csv` module will let you define the output format of your report with more flexibility than a series of manual prints.

Now you can proceed to extract all applications that share the name you passed on the command line:

```
applications = zh.application.get(output="shorten", filter={"name":
[appname] })
```


Once we have the list of applications, you can get the list of items that belong to the said application:

```
items = zh.item.get(output="count", applicationids=[x['applicationid']
for x in applications])
```

And from there, you still need to extract all the triggers that contain the given items before moving to the actual events:

```
triggers = zh.trigger.get(output="refer", itemids=[x['itemid'] for x
in items])
```

Now you can finally get the list of events that are related to the application you are interested in:

```
events = zh.event.get(triggerids=[j['triggerid'] for j in triggers])
```

Here only the event IDs are extracted. You didn't ask for a specific time period, so it's possible that a great number of events will be extracted. For every event, we'll also need to extract all related hosts, triggers, and items. To do that, let's first define a couple of helper functions to get the host, item, and trigger names:

```
def gethostname(hostid=''):
    return zh.host.get(hostids=hostid, output=['host'])[0]['host']

def getitemname(itemid=''):
    return zh.item.get(itemids=itemid, output=['name'])[0]['name']

def gettriggername(triggerid=''):
    return zh.trigger.get(triggerids=triggerid,
    expandDescription="1", output=['description'])[0]['description']
```

Finally, you can define an empty `eventstable` table and fill it with event information based on what you have extracted so far:

```
eventstable = []
triggervalues = ['OK', 'problem', 'unknown']
for e in events:
    eventid = e['eventid']
    event = zh.event.get(eventids=eventid,
        selectHosts="refer",
        selectItems="refer",
        selectTriggers="refer",
        output="extend")
    host = gethostname(event[0]['hosts'][0]['hostid'])
    item = getitemname(event[0]['items'][0]['itemid'])
```

```

        trigger = gettriggername(event[0]['triggers'][0]['triggerid'])
        clock = datetime.fromtimestamp(int(event[0]['clock'])).
        strftime(timeformat)
        value = triggervalues[int(event[0]['value'])]
        eventstable.append({"Host": host,
                           "Item": item,
                           "Trigger": trigger,
                           "Status": value,
                           "Time" : clock
                           })

```

Now that you have all the events' details, you can create the output .csv file:

```

filename = "events_" + appname + "_" + datetime.now().
strftime("%Y%m%d%H%M")
fieldnames = ['Host', 'Item', 'Trigger', 'Status', 'Time']
outfile = open(filename, 'w')
csvwriter = csv.DictWriter(outfile, delimiter=';',
fieldnames=fieldnames)
csvwriter.writerow(dict((h,h) for h in fieldnames))
for row in eventstable:
    csvwriter.writerow(row)
outfile.close()

```

The report's filename will be automatically generated based on the application you want to focus on and the time of execution. Since every event in the `eventstable` array is a dict, a `fieldnames` array is needed to tell the `csv.DictWriter` module in what order the fields should be written. Next, a column header is written before finally looping over the `eventstable` array and writing out the information we want.

There are a number of ways you can expand on this script in order to get even more useful data. Here are a few suggestions, but the list is limited only by your imagination:

- Ask for an optional time period to limit the number of events extracted
- Order events by host and trigger
- Perform some calculations to add event duration based on trigger state change
- Add acknowledge data if present

Creating graphs from data

At this point in the book, you should be familiar with Zabbix's powerful data visualization possibilities. On the frontend, you can create and visualize many kinds of graphs, maps, and charts that can help you analyze and understand item history data, trigger status change over time, IT services availability, and so on. Just as any other of Zabbix's capabilities, all of the visualization functions are also exposed through the API. You can certainly write programs to create, modify, or visualize screens, graphs, and maps, but unless you are building a custom frontend, it's quite unlikely that you'll ever need to do so.

On the other hand, it may be interesting to extract and visualize data that is otherwise too dispersed and hard to analyze through the frontend. A good example of such data is trigger dependency. You may recall from *Chapter 5, Visualizing Data*, that a trigger can depend on one or more other triggers such that it won't change to a PROBLEM state if the trigger it depends on is already in a PROBLEM state.

As useful as this feature is, there's no easy way to see at a glance the triggers that depend on other triggers, and if those triggers in turn depend on other triggers, and so on. The good news is that with the help of the Graphviz package and a couple of lines of Python code, you can create a handy visualization of all trigger dependencies.

The Graphviz suite of programs

Graphviz (<http://www.graphviz.org>) is a suite of graph visualization software utilities that enables you to create arbitrary complex graphs from especially formatted text data. The suite provides many features for data visualization and can become quite complex to use, but it is quite simple to create a basic, functional setup that you can later build on.

If you do not have it installed in your system, Graphviz is just a `yum install` command away if you are on a Red Hat Enterprise Linux system:

```
# yum install 'graphviz*
```

The program you will use to create your graphs is called `dot`. `Dot` takes a graph text description and generates the corresponding image in a number of formats. A graph description looks like this:

```
digraph G {
    main → node1 → node2;
    main → node3;
    main → end;
    node2 → node4;
    node2 → node5;
```

```

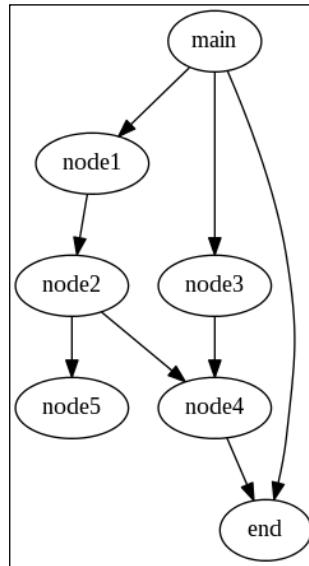
    node3 → node4;
    node4 → end;
}

```

If you put the preceding graph in a `graph.gv` file and run the following command:

```
$ dot -Tpng graph.gv -o graph.png
```

You will obtain a PNG image file that will look something like the following:



As you can see, it should be fairly simple to create a trigger-dependency graph once we have extracted the right information through the API. Let's see how we can do it.

Creating a trigger dependency graph

The following is a Python script that will extract data about trigger dependencies and output a dot language graph description that you can later feed into the dot program itself:

```

#!/usr/bin/python
from pyzabbix import ZabbixAPI
zh = ZabbixAPI("https://127.0.0.1/zabbix")
zh.login(user="Admin", password="zabbix")

def gettriggername(triggerid=''):
    return zh.trigger.get(triggerids=triggerid,
        output=['description'])[0]['description']

```

In the first part, there are no surprises, a Zabbix API session is initiated, and a simple helper function, identical to the one shown before, is defined.

```
tr = zh.trigger.get(selectDependencies="refer", expandData="1",
output="refer")
dependencies = [(t['dependencies'], t['host'], t['triggerid']) for t
in tr if t['dependencies'] != [] ]
```

The next two lines extract all triggers and their dependencies and then create a list, filtering out triggers that don't have any dependencies.

```
outfile = open('trigdeps.gv', 'w')
outfile.write('digraph TrigDeps {\n')
outfile.write('graph[rankdir=LR]\n')
outfile.write('node[fontsize=10]\n')
```

Here the first few lines of the graph are written out to the output file, setting up the graph direction, from left to right, and the font size for the nodes' labels.

```
for (deps, triggerhost, triggerid) in dependencies:
    triggername = gettriggername(triggerid)

    for d in deps:
        depname = gettriggername(d['triggerid'])
        dephost = d['host']
        edge = '"{:}\n{:}" -> "{:}\n{:}";'.format(dephost, depname,
triggerhost, triggername)
        outfile.write(edge + '\n')
```

This is the core of the script. The double for loop is necessary because a single trigger may have more than one dependency, and you want to map out all of them. For every dependency → trigger relationship found, an edge is defined in the graph file.

```
outfile.write('}\n')
outfile.close()
```

Once the script reaches the end of the list, there is nothing more to do except to close the graph description and close the output file.

If you execute the script:

```
$ chmod +x triggerdeps.py
$ ./triggerdeps.py
```

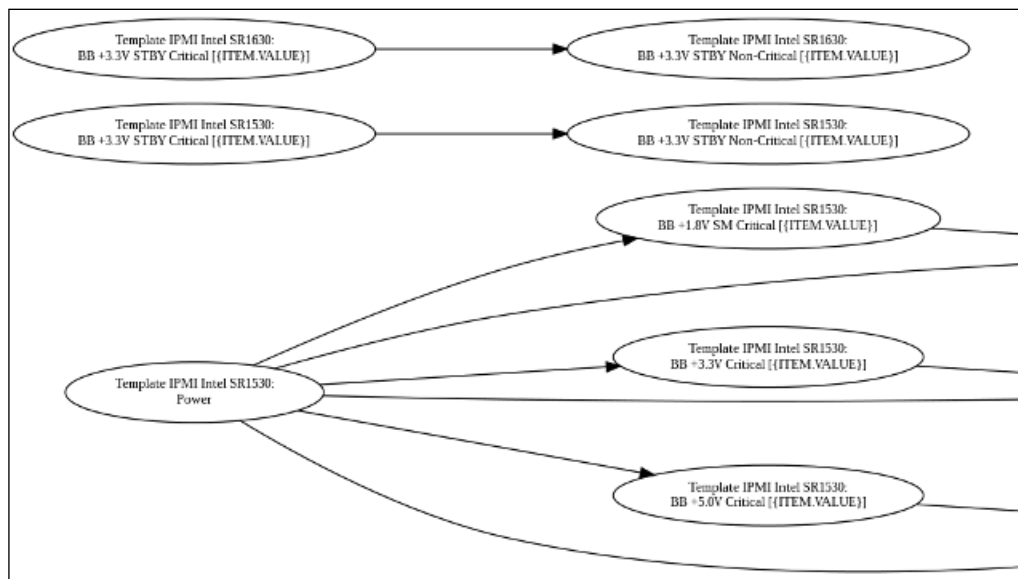
You will get a `trigdeps.gv` file that will look something like this:

```
digraph TrigDeps {
graph[rankdir=LR]
node[fontsize=10]
"Template IPMI Intel SR1630:\nPower" -> "Template IPMI Intel SR1630:\nBaseboard Temp Critical [{ITEM.VALUE}]";
"Template IPMI Intel SR1630:\nBaseboard Temp Critical [{ITEM.VALUE}]"
-> "Template IPMI Intel SR1630:\nBaseboard Temp Non-Critical [{ITEM.VALUE}]";
"Template IPMI Intel SR1630:\nPower" -> "Template IPMI Intel SR1630:\nBaseboard Temp Non-Critical [{ITEM.VALUE}]";
"Template IPMI Intel SR1630:\nPower" -> "Template IPMI Intel SR1630:\nnBB +1.05V PCH Critical [{ITEM.VALUE}]";
"Template IPMI Intel SR1630:\nnBB +1.05V PCH Critical [{ITEM.VALUE}]"
-> "Template IPMI Intel SR1630:\nnBB +1.05V PCH Non-Critical [{ITEM.VALUE}]";
"Template IPMI Intel SR1630:\nPower" -> "Template IPMI Intel SR1630:\nnBB +1.05V PCH Non-Critical [{ITEM.VALUE}]";
"Template IPMI Intel SR1630:\nPower" -> "Template IPMI Intel SR1630:\nnBB +1.1V P1 Vccp Critical [{ITEM.VALUE}]";
```

Just run it through the `dot` program in order to obtain your dependencies graphs:

```
$ dot -Tpng trigdeps.gv -o trigdeps.png
```

The resulting image will probably be quite big; the following is a close up of a sample resulting image:



Once again, there are many ways to improve the script, from improving the layout and the node shapes, to integrating the graph generating part into Python with its graphviz bindings. Moreover, you could feed the image back to a Zabbix map using the API, or you could invert the process and define trigger dependencies based on an external definition.

Summary

In this chapter, we have barely scratched the surface of what is possible once you begin playing with the powerful Zabbix API. If you have worked through the examples, you should be comfortable with the JSON-RPC protocol that is the foundation of the API. You should know how to explore the various methods and have some ideas on how to use them to make your Zabbix management tasks easier, or to further expand the system's data manipulation possibilities.

With the discussion of the API, we conclude our exploration of Zabbix's features. The next and final chapter will build upon the knowledge we have gained until now and use it to make Zabbix a more integral part of your IT infrastructure by opening communication channels with other management systems.

10

Integrating Zabbix

A monitoring system is, by definition, all about connecting and communicating with other systems. On one hand, it needs to connect to its monitored objects in order to take measurements and evaluate service status. On the other hand, it needs to be able to communicate the collected information outside of itself so that system administrators can act on the data and alarms it raised. In the previous chapters of the book, we focused mainly on the first part of the equation, namely collecting data, and always assumed that the second part, exposing data and warnings, would involve sending a series of messages to human operators. While this is certainly the most common setup, the one that will be at the core of every Zabbix deployment, it's also true that it can prove to be quite limited in a large, complex IT environment.

Every managing system has a specific, detailed view of its environment that is directly dependent on the function it must perform. Identity management systems know all about users, passwords, and permissions, while inventory systems keep track of hardware and software configurations and deployment. Trouble ticketing systems keep track of current issues with users, while monitoring systems keep track of the availability status and performance metrics of anything they can connect to. As many of these systems actually share some common data among them, whether it is user information, connection configuration or anything else, it is vital that as much of this data as possible is allowed to pass from one system to the next, without constant manual intervention on the part of the administrators.

It will be impossible for Zabbix, or any monitoring system, to come with an out-of-the-box integration with any other arbitrary system in the world. Its open source nature, clear protocol, and powerful APIs make it relatively easy to integrate your monitoring system with any other IT management tools you have deployed in your environment. This can be the subject of a book in itself, but we will try to get you started on the path of Zabbix integration by looking at one such integration possibility.

In this chapter, you will see a concrete example of integration between Zabbix and Request Tracker, the open source trouble ticket management system from Best Practical (<http://bestpractical.com/rt/>). By the end of the chapter, you will be able to do the following:

- Rely an alert to a trouble ticketing system
- Keep track of which tickets relate to which events
- Update events' acknowledgments based on the status of a ticket

There won't be any new concepts or Zabbix functionality explained here, as we'll explore some of the real-world applications made possible by the features we have already learned about in the rest of the book.

An overview of Request Tracker

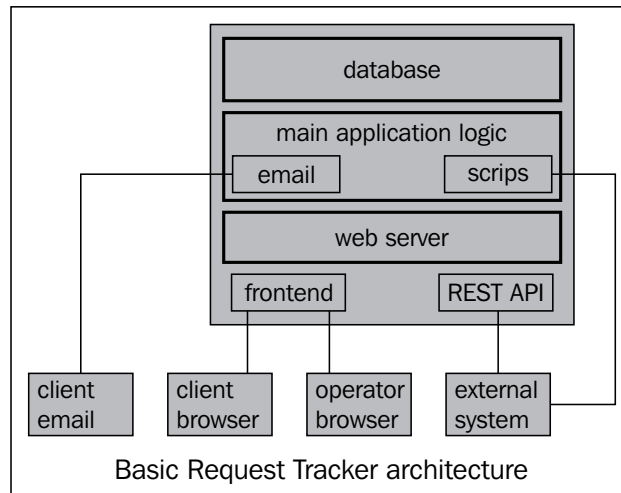
Quoting from the Best Practical website:

"RT is a battle-tested issue tracking system which thousands of organizations use for bug tracking, help desk ticketing, customer service, workflow processes, change management, network operations, youth counseling and even more. Organizations around the world have been running smoothly thanks to RT for over 10 years."

In other words, as powerful yet simple open source package it's a good candidate to showcase a Zabbix integration. This is not to say that it is the only issue tracking system that you can use with Zabbix; once the principles behind the following sample implementation are clear, you will be able to integrate any product with your monitoring system.

Request Tracker (RT) is a web application written in Perl that relies on a web server to expose its frontend and on a relational database to keep all its data on. The main means of interaction with the system is through the web interface, but it also features a powerful e-mail-parsing utility that can categorize an e-mail message, turn it into a full-fledged ticket, and keep track of the subsequent mail exchange between the user and the support staff. Closer to our interests, it also features a simple yet effective REST API, that we'll rely on in order to create and keep track of existing tickets from Zabbix. On the other hand, a powerful scripting engine that can execute custom chunks of code called **scrips**, not only allows RT to automate its internal workings and create custom workflows but also allows it to communicate with external systems using any available protocol.

The following diagram shows the basic application architecture. All the data is kept in a database, while the main application logic can interact with the outside world either through the web server or via e-mail and custom scripts.



This is not the place to cover an in-depth installation and configuration of RT, so we will assume you already have a working RT server with at least a few users and groups already set up. If you need to install RT from scratch, the procedure is quite simple and well-documented; just follow the instructions detailed here <http://www.bestpractical.com/docs/rt/4.2/README.html>. Refer to the Request Tracker website link provided earlier for further information.

Setting up RT to better integrate with Zabbix

The two basic elements of RT are tickets and queues. The function of a ticket is to keep track of the evolution of an issue. The basic lifecycle of a ticket can be summarized in the following points:

- A ticket is created with the first description of the problem
- An operator takes the ownership of the ticket and starts working on it
- The evolution of the problem is recorded in the ticket's history
- After the problem's resolution, the ticket is closed and archived

All of the ticket's metadata, from its creation date to the amount of time it took to close it, from the user who created it to the operator that worked on it, and so on, is recorded and grouped with all the other tickets' metadata in order to build statistics and calculate service-level agreements.

A queue, on the other hand, is a specific collection of tickets and a way to file new tickets under different categories. You can define different queues based on different organization departments, or different products you are providing support for, or any other criteria that can make it easier to organize tickets.

Let's see what can we do to configure RT queues and tickets so that we can transfer all the information we need to and from Zabbix, while keeping any existing functionality as a generic issue tracker as is.

Creating a custom queue for Zabbix

A nice feature of queues is that you can customize every aspect of a ticket that belongs to a specific queue, from the fields that need to be filled out to the details of the workflow. The first step is, therefore, to create a specific queue for all tickets created from a Zabbix event action. This will allow us to define specific characteristics for the corresponding Zabbix tickets.

Creating a queue is quite simple. Just go to **Admin | Queues | Create** and fill in the form. For our purpose, you don't need to specify more than a name for the queue and an optional description, as shown in the following screenshot:

The screenshot shows the 'Create a queue' form in a web interface. The form has a blue header with the title 'Create a queue' and a 'New ticket in' button. Below the header, there are tabs for 'General' and 'Search...'. The form fields are as follows:

- Queue Name:
- Description:
- Lifecycle:
- Subject Tag:
- Reply Address:
- Comment Address:
- Priority starts at:
- Over time, priority moves toward:
- Requests should be due in: days.
- ☒ Enabled (Unchecking this box disables this queue)
-

After the queue is created, you will be able to configure it further by going to **Admin** | **Queues** | **Select** and choosing the Zabbix queue. At the very least, you should grant the user and staff rights to a user group or to some specific users so that your IT staff can work on the tickets created by Zabbix. You will also want to create some custom fields, as we will see in a couple of paragraphs.

First, let's move on to look at what parts of a ticket are most interesting from an integrator's point of view.

Customizing tickets – the links section

Keeping in mind our goal to integrate Zabbix actions and events with RT, the **Links** section of a ticket is of particular interest to us. As the name suggests, you can define links to other tickets as dependencies or even define links to other systems as further referrals. You can insert useful links during ticket creation or while editing it.

Link ticket #3 New ticket in General Search...

Display History Basics People Dates **Links** Jumbo Reminders Actions

☆

^ Edit Links

Current Links	New Links
Depends on:	<i>Enter tickets or URIs to link tickets to. Separate multiple entries with spaces. You may enter links to Articles as "a:###", where ### represents the number of the Article.</i>
Depended on by:	Depends on: <input type="text"/>
Parents:	Depended on by: <input type="text"/>
Children:	Parents: <input type="text"/>
Refers to:	Children: <input type="text"/>
Referred to by:	Refers to: <input type="text"/>
<i>(Check box to delete)</i>	Referred to by: <input type="text"/>

As you probably already imagined, we'll be relying on the **Refers to:** link field to link back to the Zabbix event that created the ticket. As we will see in the following pages, the event's acknowledged field will in turn show a link to the corresponding ticket so that you can move easily from one platform to the other in order to keep track of what's happening.

Customizing tickets – ticket priority

Another interesting field in the **Basics** section of a ticket is the ticket's priority. It's an integer value that can go from 0 to 100, and it's quite useful to sort tickets depending on their severity level. There is no official mapping of RT priority and some more descriptive severity levels such as those used by the Zabbix triggers. This means that if you want to preserve information about trigger severity in the ticket, you have two choices:

- Ignore the ticket's priority and create a custom field that shows the trigger severity as a label
- Map the trigger severity values to the ticket's priority values as a convention, and refer to the mapping while creating tickets

The only advantage of the first option is that the single ticket will be easily readable, and you will immediately know about the severity of the corresponding trigger. On the other hand, the second option will allow you to sort your tickets by priority and act first on the more important or pressing issues with a more streamlined workflow.

While creating a ticket from Zabbix, our suggestion is, therefore, to set ticket priorities based on the following mapping:

Trigger severity label	Trigger severity value	Ticket priority value
Not classified	0	0
Information	1	20
Warning	2	40
Average	3	60
High	4	80
Disaster	5	100

There is nothing to configure either on the Zabbix or RT side. This mapping will use the full range of priority values so that your Zabbix tickets will be correctly sorted not only in their specific queue, but also anywhere in RT.

Customizing tickets – the custom fields

As we have seen in *Chapter 6, Managing Alerts*, a Zabbix action can access a great number of macros and thus expose a lot of information about the event that generated it. While it makes perfect sense to just format this information in a readable manner while sending e-mails, with the availability of custom fields for RT tickets, it makes less sense to limit all of the event details just to the ticket description.

In fact, one great advantage of custom fields is that they are searchable and filterable just like any other native ticket field. This means that if you put the host related to a ticket event in a custom field, you'll be able to search all tickets belonging to the said host for reporting purposes, or assign a host's specific tickets to a particular user, and so on. So, let's go ahead and create a couple of custom fields for the tickets in the Zabbix queue that will contain information, which we'll find useful later on. Go to **Admin | Custom Fields | Create** and create a **Hosts** custom field as shown in the following screenshot:

Custom Fields ▾ Basics Group Rights User Rights Applies to

Name

Description

Type

Applies to

Validation

Link values to

Include page

☒ Enabled (Unchecking this box disables this custom field)

[Save Changes](#)

Be sure to select **Enter multiple values** as the field type. This will allow us to specify more than a single host for those complex triggers that reference items from different hosts.

Speaking of triggers and items, you can follow the same procedure to create other custom fields for the trigger name and item names or keys. After you are done, you will need to assign these fields to the tickets in the Zabbix queue. Select the Zabbix queue by going to **Admin | Queues | Select** and for the **Tickets** form, go to **Custom fields | Tickets**. Select the fields that you wish to assign to your tickets:

Selected Custom Fields

Unselected Custom Fields

<input type="checkbox"/>	#	Name	Added	Type	Pattern
<input checked="" type="checkbox"/>	2	Hosts		Enter multiple values	
<input checked="" type="checkbox"/>	4	Items		Enter multiple values	
<input checked="" type="checkbox"/>	3	Trigger		Enter one value	

Submit

After you are done, you will see the following fields in every ticket of the Zabbix queue:

Create a new ticket

New ticket in

General

Search...

Basics

Details

Basics

Queue: Zabbix

Status: new

Owner: Nobody in particular

Custom Fields

Hosts:

Enter multiple values

Items:

Enter multiple values

Trigger:

Enter one value

Create a new ticket

Depending on your needs, you can create as many custom fields as you want for trigger and event acknowledged history, host's IP interfaces, custom macros, and so on. You will be able to search for any of them, and for the three shown earlier, you can do so by selecting the Zabbix queue in the search page of the RT frontend. At the bottom of the search form, you can see the newly created fields, just as expected:

The screenshot displays the 'Query Builder' interface. At the top, there's a header with 'New ticket in' and a 'General' dropdown. Below this is a navigation bar with 'Edit Search', 'Advanced', 'Show Results', 'Bulk Update', 'Chart', and 'Feeds'. The main area is divided into two columns. The left column, titled 'Add Criteria', contains a list of fields with dropdown menus for operators and input fields for values. The fields include 'id', 'Subject', 'Queue', 'Status', 'Owner', 'Requestor Email', 'Owner Group', 'Created', 'Time Worked', 'Priority', 'Child', 'Hosts', 'Items', and 'Trigger'. The right column, titled 'Current search', shows a list of active search criteria, currently containing 'Queue = 'Zabbix''. Below this list are buttons for adding, removing, and deleting criteria, as well as an 'And/Or' button. At the bottom right, there's a 'Saved searches' section with a 'Privacy' dropdown, a 'Description' field, and a 'Save' button. There's also a 'Load saved search' dropdown and a 'Load' button.

Connecting to the Request Tracker API

RT exposes a REST-type API that relies directly on the HTTP protocol to handle requests and responses. This means that the API is easily tested and explored by using tools such as `wget` and `netcat`. Let's do that to get a feel of how it works before introducing the Python library that we'll use for the rest of the chapter.

The base URL for the RT API is located at `../REST/1.0/` after the base URL of the Request Tracker itself. This means that if your base URL is `http://your.domain.com/rt`, the API will be accessible at `http://your.domain.com/rt/REST/1.0/`.

If you try to connect to it, you should get a message asking for credentials (some response headers are removed to improve readability):

```
$ ncat example.com 80
GET /rt/REST/1.0/ HTTP/1.1
Host: example.com

HTTP/1.1 200 OK
[...]
Content-Type: text/plain; charset=utf-8

22
RT/4.2.0 401 Credentials required
```

The API doesn't have a special authentication mechanism separated from the rest of the application, so the best way to authenticate is to get a session cookie from the main login form and use it for each API request. To get the cookie, let's use `wget`:

```
$ wget --keep-session-cookies --save-cookies cookies.txt --post-data
'user=root&pass=password' http://example.com/rt/
```

The command that will save the session cookie in the `cookies.txt` file is as follows:

```
$ cat cookies.txt
# HTTP cookie file.
# Generated by Wget on 2013-11-02 10:16:58.
# Edit at your own risk.

localhost FALSE /rt FALSE 0 RT_SID_example.com.80
2bb04e679236e58b406b1e554a47af43
```

Now that we have a valid session cookie, we can issue requests through the API. Here is the `GET` request for the general queue:

```
$ ncat localhost 80
GET /rt/REST/1.0/queue/1 HTTP/1.1
Host: localhost
Cookie: RT_SID_example.com.80=2bb04e679236e58b406b1e554a47af43

HTTP/1.1 200 OK
[...]
Content-Type: text/plain; charset=utf-8

RT/4.2.0 200 Ok
```

```
id: queue/1
Name: General
Description: The default queue
CorrespondAddress:
CommentAddress:
InitialPriority: 0
FinalPriority: 0
DefaultDueIn: 0
```

As you can see, the API is quite easy to interact with without any special encoding or decoding. For our purposes, however, it is even easier to use a library that will spare us the burden of parsing each HTTP request. `Rtkit` is a Python 2.x library that makes it even easier to connect to the API from within a Python program, allowing us to send requests and get responses using native Python data structures. The installation is very simple using `pip`:

```
$ pip install python-rtkit
```

Once installed, the library will be available by importing various `Rtkit` modules. Let's see the same preceding interaction (authenticating and requesting the general queue) from within a Python 2.x session:

```
$ python2
Python 2.7.5 (default, Sep  6 2013, 09:55:21)
[GCC 4.8.1 20130725 (prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from rtkit.resource import RTResource
>>> from rtkit.authenticators import CookieAuthenticator
>>> from rtkit.errors import RTResourceError
>>>
>>> res = RTResource('http://localhost/rt/REST/1.0/', 'root', 'password',
CookieAuthenticator)
>>>
>>> response = res.get(path='queue/1')
>>> type(response)
<class 'rtkit.resource.RTResponse'>
>>> type(response.parsed)
<type 'list'>
>>> response.parsed
[[('id', 'queue/1'), ('Name', 'General'), ('Description', 'The
default queue'), ('CorrespondAddress', ''), ('CommentAddress', ''),
('InitialPriority', '0'), ('FinalPriority', '0'), ('DefaultDueIn', '0')]]
```

As you can see, a response is parsed into a list of tuples with all the attributes of an RT object.

Now that we have a custom queue and custom fields for Zabbix tickets, we are able to interact with the API through the Python code, and the setup process on the RT's side is complete. We are ready to actually integrate the Zabbix actions and the RT tickets.

Setting up Zabbix to integrate with Request Tracker

Our goal is to define a Zabbix action step that when executed will:

- Create a ticket with all the relevant event information
- Link the ticket back to the Zabbix event that generated it
- Acknowledge the event with a link to the ticket just created

While the first point can be covered with a simple e-mail action to RT, we need some custom code to take care of the other two. The best way to do this is to define a new media type in Zabbix as a custom alert script. The script will:

- Take the action message
- Parse it to extract relevant information
- Create a ticket with all custom fields and link the referrals filled out
- Get back the ticket ID
- Write a link to the created ticket in the event's acknowledgment field

Before actually writing the script, let's create the media type and link it to a user (you can assign the media type to any user you want; the custom `rt_tickets` user will be used here).

CONFIGURATION OF MEDIA TYPES

Media

Description

Type

Script name

Enabled ☒

Zabbix 2.0.8 Copyright 2001-2013 by Zabbix SIA | Connected as 'Admin' from 'ZBX Main'

While linking the media type to the user, use the RT base URL in the **Send to** field, so you won't need to define it statically in the script:

New media

Type

Send to

When active

☒ Not classified

☒ Information

☒ Warning

Use if severity ☒ Average

☒ High

☒ Disaster

Status

Once saved, you'll see all relevant information at a glance in the Media tab. Just after the URL address, you'll find the notification periods for the media, and after that a six-letter code that shows the active severity levels. If you disabled any of them, the corresponding letter would be gray.

CONFIGURATION OF USER						
User	Media	Permissions				
	Media	<input type="checkbox"/>	Request Tracker	http://localhost/rt/	1-7,00:00-24:00	NIWAHD Enabled Edit
Add Delete selected						
<div>Save Delete Cancel</div>						
Zabbix 2.0.8 Copyright 2001-2013 by Zabbix SIA Connected as 'Admin' from 'ZBX Main'						

Now let's create an action step that will send a message to our `rt_tickets` user through the custom media type. Needless to say, the `rt_tickets` user won't receive any actual message as the alert script will actually create a RT ticket, but all of this is completely transparent from the point of view of a Zabbix action. You can put any information you want in the message body, but, at the very least, you should specify the trigger name in the subject and the event ID, severity, hosts, and items in the body so that the script will parse them and fill them in the appropriate ticket fields:

Operation details	Step	From	<input type="text" value="1"/>						
		To	<input type="text" value="1"/> (0 - infinitely)						
		Step duration	<input type="text" value="0"/> (minimum 60 seconds, 0 - use action default)						
	Operation type	<input type="text" value="Send message"/>							
	Send to User groups	<table border="1"> <thead> <tr> <th>User group</th> <th>Action</th> </tr> </thead> <tbody> <tr> <td>Add</td> <td></td> </tr> </tbody> </table>		User group	Action	Add			
	User group	Action							
	Add								
	Send to Users	<table border="1"> <thead> <tr> <th>User</th> <th>Action</th> </tr> </thead> <tbody> <tr> <td>rt_tickets</td> <td>Remove</td> </tr> <tr> <td>Add</td> <td></td> </tr> </tbody> </table>		User	Action	rt_tickets	Remove	Add	
	User	Action							
	rt_tickets	Remove							
Add									
Send only to	<input type="text" value="Request Tracker"/>								
Default message	<input type="checkbox"/>								
Subject	<input data-bbox="662 871 1239 905" type="text" value="{TRIGGER.NAME}"/>								
Message	<div style="border: 1px solid orange; padding: 5px;"> <p>Event: {TRIGGER.ID}&eventid={EVENT.ID} Host: {HOST.NAME} Item: {ITEM.NAME}</p> <p>Trigger: {TRIGGER.NAME} Trigger status: {TRIGGER.STATUS} Trigger severity: {TRIGGER.SEVERITY} Trigger URL: {TRIGGER.URL}</p> <p>Item values:</p> <ol style="list-style-type: none"> 1. {ITEM.NAME1} ({HOST.NAME1}-{ITEM.KEY1}): {ITEM.VALUE1} 2. {ITEM.NAME2} ({HOST.NAME2}-{ITEM.KEY2}): {ITEM.VALUE2} 3. {ITEM.NAME3} ({HOST.NAME3}-{ITEM.KEY3}): {ITEM.VALUE3} </div>								
Conditions	<table border="1"> <thead> <tr> <th>Label</th> <th>Name</th> <th>Action</th> </tr> </thead> <tbody> </tbody> </table>		Label	Name	Action				
Label	Name	Action							

We are now ready to actually write the script and use it to create tickets from Zabbix.

Creating RT tickets from the Zabbix events

Zabbix will search for custom alert scripts in the directory specified by `AlertScriptsPath` in the `zabbix_server.conf` file. In the case of a default install, this would be `/usr/local/share/zabbix/alertscripts/`.

This is where we will put our script called `rt_mkticket.py`. The Zabbix action we configured earlier will call this script with the following three arguments, in order:

- recipient
- subject
- message

As we have seen, the content of the subject and the message are defined in the action operation and depend on the specifics of the event triggering action. The recipient is defined in the media type configuration of the user receiving the message, and it is usually an e-mail address. In our case, it will be the base URL of our Request Tracker installation.

So let's start the script by importing the relevant libraries and parsing the arguments:

```
#!/usr/bin/python2

from pyzabbix import ZabbixAPI

from rtkit.resource import RTResource

from rtkit.authenticators import CookieAuthenticator

from rtkit.errors import RTResourceError

import sys

import re

rt_url = sys.argv[1]
rt_api = rt_url + 'REST/1.0/'

trigger_name = sys.argv[2]

message= sys.argv[3]
```

Now we need to extract at least the event URL, the trigger severity, a list of host names, and a list of item names from the message. To do this, we will use the powerful regular expression functions of Python:

```
event_id = re.findall(r'^Event: (.+)$', message, re.MULTILINE)[0]

severity = re.findall(r'^Trigger severity: (.+)$', message,
re.MULTILINE)[0]
hosts = re.findall(r'^Host: (.+)$', message, re.MULTILINE)

items = re.findall(r'^Item: (.+)$', message, re.MULTILINE)
lines = re.findall(r'^(! (Host:|Event:|Item:|Trigger severity:))
(.*)$', message, re.MULTILINE)

desc = '\n'.join([y for (x, y) in lines])
```

While the event ID has to be unique, a trigger can reference more than one item, and thus more than one host. The preceding code will match any line beginning with `Host:` to build a list of hosts. In the preceding action message, we just put one `Host: {HOST.NAME}` line for readability purposes, but your actual template can contain more than one (just remember to use `{HOST.NAME1}`, `{HOST.NAME2}`, `{HOST.NAME3}`, and so on, or you'll end up with the same host value repeatedly). Of course, the same goes for item names. The rest of the message is then extracted with the opposite of the regexps used before and joined back in a single multiline string.

Now, the macro we used for trigger severity is `{TRIGGER.SEVERITY}`. This means that it will be substituted by a string description and not a numerical value. So, let's define a simple dictionary with severity labels and RT ticket priority values mapped, as explained earlier in the chapter:

```
priorities = {

    'Not classified': 0,

    'Information': 20,

    'Warning': 40,

    'Average': 60,

    'High': 80,

    'Disaster': 100 }
```


We also need to know in advance the name of the queue we are creating the ticket in, or better yet, its ID number:

```
queue_id = 3
```

Now that we have everything we need, we can proceed and build the request to create a new ticket and then send it over to Request Tracker.

```
ticket_content = {
    'content': {
        'Queue': queue_id,
        'Subject': trigger_name,
        'Text': desc,
        'Priority': priorities[severity],
        'CF.{Hosts}': ','.join(hosts),
        'CF.{Items}': ','.join(items),
        'CF.{Trigger}': trigger_name
    }
}

links = {
    'content': {
        'RefersTo': event_url
    }
}
```

First, we create two dictionaries, one for the main ticket content, and the second for the links section, which must be edited separately.

Then we get to the main part of the script: first we log in to the RT API (be sure to use your actual username and password credentials!), then we create a new ticket, and then we get the ticket ID and input the link to the Zabbix event page:

```
rt = RTResource(rt_api, 'root', 'password', CookieAuthenticator)

ticket = rt.post(path='ticket/new', payload=ticket_content,)

(label,ticket_id) = ticket.parsed[0][0]

refers = rt.post(path=ticket_id + '/links', payload=links,)
```

We are almost done. All that is left to do is to acknowledge the Zabbix event with a link back to the ticket we just created:

```
event_id = re.findall(r'eventid=(\d+)', event_url)[0]

ticket_url = rt_url + 'Ticket/Display.html?id=' + ticket_id.split('/')[1]

print(ticket_url)

zh = ZabbixAPI('http://localhost/zabbix')

zh.login(user='Admin', password='zabbix')

ack_message = 'Ticket created.\n' + ticket_url

zh.event.acknowledge(eventids=event_id, message=ack_message)
```

This preceding code is fairly straightforward. After extracting the `eventid` value and creating the URL to the ticket, we connect to the Zabbix API and edit the `acknowledge` field of the event, effectively closing the circle.

Now that the script is complete, remember to give ownership to the `zabbix` user and set the executable bit on it:

```
$ chown zabbix rt_mkticket.py
$ chmod +x rt_mkticket.py
```

The next time the action condition you defined in your system returns `true`, and the action operation is carried out, the script will be executed with the parameters we've seen before. A ticket will be created with a link back to the event, and the event itself will be acknowledged with a link to the ticket.

Here is an example event. The link in the acknowledgement field corresponds to the URL of the ticket:

The screenshot displays the Zabbix web interface. At the top, the 'ZABBIX' logo is on the left, and navigation links like 'Help', 'Get support', 'Print', 'Profile', and 'Logout' are on the right. Below the logo, a menu bar includes 'Monitoring', 'Inventory', 'Reports', 'Configuration', and 'Administration'. The 'Administration' tab is selected, showing 'ZBX main' and 'Current node: ZBX Main'. A search bar is also present.

The main content area shows a breadcrumb trail: 'History: Latest events > Event details > Latest events > Event details > Latest events'. Below this, a message states: 'EVENTS: "Zabbix agent on ZBX Main is unreachable for 2 minutes"'. The interface is divided into two main sections: 'Event source details' and 'Event details' on the left, and 'Acknowledges' and 'Message actions' on the right.

Event source details:

- Host: [ZBX Main](#)
- Trigger: Zabbix agent on ZBX Main is unreachable for 2 minutes
- Severity: Average
- Expression: `{ZBX Main:agent.ping nodata(2m)}=1`
- Event generation: Normal
- Disabled: No

Event details:

- Event: Zabbix agent on ZBX Main is unreachable for 2 minutes
- Time: 04 Nov 2013 01:39:30
- Acknowledged: Yes (1)

Acknowledges:

Time	User	Comments
04 Nov 2013 01:39:41	Admin	Ticket created. http://localhost/r/Ticket/Display.html?id=28

Message actions:

Time	Type	Status	Retries left	Recipient(s)	Mes
Step: 1	Request Tracker	scnt		http://localhost/r/	Subj
Time:					Zabb
04 Nov 2013 01:39:32					Mes
					Even
					trigge
					Host:
					Item:

And here is the corresponding ticket. The **Refers to:** field contains a clickable link to the event shown earlier, while the **Custom Fields** section reports host, item, and trigger information, just as expected:

#27: Zabbix agent on Delta is unreachable for 2 minutes
New ticket in
General
Search...

Display
History
Basics
People
Dates
Links
Jumbo
Reminders
Actions
☆

Ticket metadata

The Basics

Id: 27
Status: new
Priority: 60/
Queue: Zabbix

Custom Fields

Hosts: Delta
Items: Agent ping
Zabbix agent on
Trigger: Delta is unreachable
for 2 minutes

People

Owner: Nobody in particular
Requestors:
Cc:
AdminCc:

Reminders

New reminder:
Subject:
Owner: root (Enoch Root)
Due:

Save

Dates

Created: Sun Nov 03 19:39:09 2013
Starts: Not set
Started: Not set
Last Contact: Not set
Due: Not set
Closed: Not set
Updated: Sun Nov 03 19:39:09 2013 by root (Enoch Root)

Links

Depends on: (Create)
Depended on by: (Create)
Parents: (Create)
Children: (Create)
Refers to: (Create)

• http://localhost/zabbix/tr_events.php?triggerid=9909900000013590

The script, in much the same way as those explained in *Chapter 9, Extending Zabbix*, is little more than a proof of concept, with as much focus on readability and ease of explanation as on pure functionality. Be sure to add as many condition checks and error-reporting functions as possible if you want to use it in a production environment.

Summary

We have finally reached the end of our journey to mastering the Zabbix monitoring system. In the course of the book, you have learned how to plan and implement the general monitoring architecture; how to create flexible and effective items, triggers, and actions; and how to best visualize your data. You have also learned how to implement custom agents by understanding the Zabbix protocol and how to write the code to manipulate every aspect of Zabbix through its API. In this chapter, we have barely scratched the surface of what's possible once you start taking advantage of what you now know about Zabbix to integrate it better with your IT infrastructure. Many more integration possibilities exist, from getting and updating users and groups through an identity management system to getting inventory information through an asset management system, or vice versa feeding inventory information to a CMDB database, and much more. Our hope is that by following the steps necessary to integrate Zabbix with a trouble ticket management system, you have learned how to prepare two systems such that they can share and exchange data and how to use each system's API in a coordinated manner in order to get the systems to talk to each other.

Our hope is that with the skills you just learned, you will be able to bring out the full potential of the Zabbix monitoring system and make it a central asset of your IT infrastructure. In doing so, your time and effort will be repaid tenfold.

Index

Symbols

`{EVENT.DATE}` macros 224
`{EVENT.TIME}` macros 224
`{HOST.CONN}` macros 236
`{HOST.DNS}` macros 236
`{HOST.HOST}` macros 236
`{HOST.IP}` macros 236
`{HOST.NAME}` macros 236
`{INVENTORY.SERIALNO.A}` macros 224
`{NODE.ID}` macros 224
`{NODE.NAME}` macros 224

A

action

- about 222
- action conditions 222
- action definition 222
- action operations 223
- defining 223, 224

action conditions

- defining 225, 226

action definition

- `{EVENT.DATE}` macros 224
- `{EVENT.TIME}` macros 224
- friends macros 224
- `{INVENTORY.SERIALNO.A}` macros 224
- `{NODE.ID}` macros 224
- `{NODE.NAME}` macros 224

action operations

- aspects 226
- media step 228, 229
- message step 228, 229
- remote commands 230

- selecting 226

- steps 227, 228

active property 296

active Zabbix proxy 63-65

aggregated items 164-166

architecture, RT 309

architecture, Zabbix

- for large environment 12, 13

- RDBMS server 12

- types 13

- web server 12

- Zabbix server 12

architecture, Zabbix proxy 56, 57

Auto-login flag 204

avg() function 165, 197

C

calculated items 164, 167

CentOS 22

check_ora_sendtrap

- wrapper script, coding for 266, 267

clock property

- using 272

configuration component 33

configuration, Corosync 89

configuration files, UnixODBC 133, 134

configuration, HTTPD 88-95

configuration, IT services 206, 208

configuration, JMX 141-143

configuration, Pacemaker

- testing 110, 111

configuration, PostgreSQL 109

configuration, web interface 42-47

configuration, Zabbix Server

performing 20, 21
for high availability 96, 97

configure command 134

Corosync

configuring 89

count function 165

curve fitting process 53

custom graphs

about 171
analyzing 174-178
exploded graphs 177
normal graphs 177
pie graphs 177
stacked graphs 177

custom media script 229

D

data

exporting 298
graphs, creating from 302
sending to Zabbix server, new script
used 265, 266
sending, zabbix_sender utility used 264
visualizing, through maps 182-185
visualizing, through slide show 201, 202

database

considerations 30, 32
high availability, implementing for 98, 99
installing 21-23
sizing 32-35

database drivers

installing 128

database management systems (DBMS) 127

database monitoring

ODBC 127
with Zabbix 126

database monitor items 135, 136

data flow

for Zabbix item 122-124
monitoring, with Zabbix node 67
monitoring, with Zabbix proxy 63-65
overview 125

Data Source Name (DSN) 127

date function 215

DBforBIX

URL 281

dedicated script server

pros and cons 267

delete_history() function 35

delete partition command 36

discovery rules, Zabbix

filesystems types 248
network interfaces 248
SNMP OIDs 248

**Distributed Replicated Block Device. *See*
DRBD**

downtime

scheduled downtime 82
unscheduled downtime 82

DRBD

about 86
configuration, enabling 107
creating, on LVM partition 103, 104
features 86, 87
filesystem, creating on 105, 106
filesystem, replicating with 86, 87
integrating, with Pacemaker 107
networking consideration 115-117
online verification, enabling 114
optimizations 112
performance 112
primary device, defining in 105
resource, enabling in 104, 105
setting, over LVM 102
synchronization 112, 113

dynamic elements, screens

about 199, 200
graphs 199
plain text 199
simple graphs 199

E

elements, maps

about 191-193
selecting 194, 195

entities

adding, to templates 234, 235

Estimate Time to Repair (ETR) 112

- event component** 33
- execquery() function** 266
- external checks**
 - about 254
 - consideration 260
 - example 255-258
 - external script, location 255
 - external script, working 258, 259
 - script writing rules 260
- external script**
 - location 255
 - working 258, 259

F

- failover**
 - automating, with resource manager 86
- filesystem**
 - creating, on DRBD 105, 106
 - replicating, with DRBD 86, 87
 - types 249
- friends macros** 224
- function**
 - selecting 213

G

- get command** 145
- gethost function** 267
- get method** 298
- getquery function** 267
- graph.gv file** 303
- graphs**
 - 3D View parameter 180
 - about 170
 - Colour parameter 181
 - creating, from data 302
 - custom graphs 171-178
 - Draw style parameter 181
 - Function parameter 181
 - Graph type parameter 179
 - Height parameter 179
 - Name parameter 179
 - Percentile line (left/right) parameter 180
 - properties 179-181
 - Show legend parameter 179

- Show triggers parameter 180
- Show working time parameter 180
- simple graphs 171-173
- Sort order parameter 180
- Type parameter 180
- Width parameter 179
- Y axis MIN/MAX value parameter 180
- Y axis side parameter 181

Graphviz

- about 302, 303
- URL 302

- grpavg function** 165

- grpmax function** 165

- grpmin function** 165

- grpsum function** 165

H

high availability

- about 82
- considerations 84, 85
- failover, automating with
 - resource manager 86
- implementing, for database 98, 99
- implementing, on web server 87
- Zabbix Server, configuring 96, 97

history component 33

host entities

- vs, template entities 235

hosts

- discovering 244-248
- redistributing, to proxy 294, 295
- templates, linking to 242

housekeeping

- about 35
- considerations 35-41

HTTPD

- configuring 88-95

I

INSERT statement 38

installation, database 21-23

installation, database drivers 128

installation, package 19

installation, PostgreSQL 9.2 22, 23

- installation, PyZabbix** 288
- installation, web interface** 41
- installation, Zabbix**
 - performing 14, 15
 - prerequisites 16
- installation, Zabbix Java gateway** 139-141
- iODBC** 127
- items**
 - selecting 213
- IT service level**
 - understanding 83
- IT services**
 - about 205
 - configuring 206, 208

J

- Java**
 - Zabbix sender protocol, implementing in 276, 277
- JMX**
 - configuring 141-143
 - considerations 144
 - issues 144
 - monitoring 137
 - security aspects 138, 139
 - Zabbix Java gateway, installing 139-141
- JMX keys** 143

L

- large environment** 10, 11
- last() function** 165, 197
- ldd command** 129
- logical operation** 217-220
- Logical Volume (LV)** 101
- logical volume manager.** *See* LVM
- low-level discovery rules** 248-251
- LVM**
 - about 100, 101
 - configuring 108
 - DRBD, setting over 102
- LVM2**
 - about 100
 - basic concepts 100
- LVM partition**
 - DRBD, creating on 103, 104

M

- macros**
 - about 188-190
 - URL, documentation 224
 - user defined macros 241
 - using 236-241
 - working with 195-197
- Management Information Base (MIB)** 145
- maps**
 - about 170
 - data, visualizing through 182-185
 - elements 191-193
 - elements, selecting 194, 195
 - macros 188-190
 - macros, working with 195-197
 - URLs 188-190
 - Zabbix Map, creating 186-188
- max() function** 165, 197
- Maximum Transmission Unit (MTU)** 115
- MBean** 143
- Mean Time To Recovery (MTTR)** 83
- min() function** 165, 197
- monitored by property** 294
- MySQL ODBC drivers** 128, 129

N

- nesting templates** 242, 243
- network configuration** 109
- network interfaces** 249
- new script**
 - used, for sending data to Zabbix server 265, 266
- node.csv file** 292
- nodes**
 - templates, synchronizing across 291-293

O

- observer effect** 48
- ODBC**
 - about 127
 - database drivers, installing 128
 - database monitor items 135, 136
 - iODBC 127
 - MySQL drivers 128, 129

- Oracle drivers 131, 132
- PostgreSQL drivers 130, 131
- SQL query, considerations 136
- UnixODBC 127
- UnixODBC configuration files 133
- Zabbix, compiling 134
- OID 145**
- online verification**
 - enabling 114
- Oracle ODBC drivers 131, 132**
- P**
- Pacemaker**
 - about 90
 - configuration, testing 110, 111
 - configuring 110
 - DRBD configuration, enabling 107
 - DRBD, integrating with 107
 - features 86
 - HTTPD, configuring 92-95
 - LVM configuration 108
 - network configuration 109
 - PostgreSQL, configuring 109
 - Quorum 91
 - stickiness concepts 92
 - URL, for downloading 86
- package**
 - creating 19
 - installing 19
- partitioning**
 - about 36
 - benefits 36
- passive Zabbix proxy 63-65**
- period property 296**
- perl trap handler 153-158**
- Physical Volume (PV) 100**
- planned downtime. *See* scheduled downtime**
- point-in-time recovery (PITR) 99**
- PostgreSQL**
 - clustering 100
 - configuring 109
- PostgreSQL 9.2**
 - installing 22, 23
 - URL 22

- PostgreSQL ODBC drivers**
 - about 130, 131
 - URL 130
- primary device**
 - defining, in DRBD 105
- proxy**
 - hosts, redistributing to 294, 295
- proxy.update method 295**
- Python**
 - Zabbix sender protocol, implementing in 278, 279
- PyZabbix**
 - installing 288
 - URL 288
 - using 288-291
- Q**
- queues, RT**
 - about 310
 - creating 310, 311
- Quorum 91**
- R**
- Recovery Time Objective (RTO) 112**
- Red Hat 22**
- remote commands 230**
- Request Tracker. *See* RT**
- resource**
 - enabling, in DRBD 104, 105
- resource manager**
 - failover, automating with 86
- RT**
 - about 308, 309
 - architecture 309
 - queues, creating 310, 311
 - setting up, to integrate with Zabbix 309
- RT API**
 - connecting to 315-318
- S**
- scheduled downtime**
 - vs, unscheduled downtime 82
- screens**
 - creating 198, 199

- dynamic elements 199, 200
- elements, adding 198
- visualizing through 197
- scrips** 308
- script writing rules** 260
- Secure Shell.** *See* SSH
- security, Zabbix**
 - network isolation 76
 - tunnels 76
 - VPN 79
 - without network configuration 74, 75
- send() function** 266
- service level agreement (SLA)** 83
- session riding** 188
- set command** 145
- severity** 216
- severity property** 296
- shell command** 261
- Shoot The Other Node In The Head.** *See* STONITH
- Siege**
 - URL 51
- simple graphs**
 - about 171
 - analyzing 171-173
- Simple Network Monitoring Protocol.** *See* SNMP
- slide show**
 - customization 204, 205
 - on big display 202
 - on big display, considerations 203
 - visualizing through 201, 202
- Snapshot Logical Volume (SLV)** 101
- SNMP** 145, 253
- SNMP monitoring**
 - about 145-148
 - SNMP queries 148-151
 - SNMP traps 151, 152
- SNMP OIDs** 249
- SNMP queries** 148-151
- snmptrapd process** 152, 153
- SNMP traps**
 - about 151, 152
 - perl trap handler 153-158
 - snmptrapd process 152, 153
- snmpwalk command** 151
- SourceForge**
 - URL 267
- SQL query**
 - considerations 136
- SSH** 76, 77
- start() function** 26
- steps, action operations** 227, 228
- stickiness** 92
- STONITH** 90
- stop() function** 27
- stunnel** 78, 79
- sudo checkinstall command** 19
- sum function** 165
- switchover.** *See* failover
- synchronization, DRBD** 112, 113

T

- tabular data**
 - extracting 299-301
- template entities**
 - vs, host entities 235
- Template OS Linux** 174
- templates**
 - about 234, 235
 - combining 244
 - creating 234
 - entities, adding to 234, 235
 - linking, to hosts 242
 - low-level discovery rules 248-251
 - nesting 242, 243
 - synchronizing, across nodes 291-293
- tickets, RT**
 - about 310
 - creating, from Zabbix 322-327
 - custom fields, customizing 313-315
 - links section, customizing 311
 - ticket priority, customizing 312
- time function** 215
- trends component** 33
- trigger dependencies**
 - managing 221
- trigger dependency graph**
 - creating 303-306

trigger expression

- about 212
- components 212
- function, selecting 213
- items, selecting 213
- logical operation 217-220
- severity 216

trigger function

- absolute value, selecting 216
- #num argument, selecting 214
- percentage, selecting 216
- sec argument, selecting 214

tunnels

- about 76
- SSH 76, 77
- stunnel 78

U

unexpected downtime. *See* **unscheduled downtime**

UnixODBC

- about 127
- configuration files 133
- URL 128

unscheduled downtime

- vs, scheduled downtime 82

URLs 188-190

user defined macros 241

user parameter

- about 261
- considerations 263
- flexibility 261, 262

users.csv file 295

users, Zabbix API

- adding 295-298
- updating 295-298

V

Values per seconds (vps) 62

verbose mode 133

Volume Group (VG) 101

VPN 79

W

web interface

- configuration 42-47
- installing 41

web pages

- authenticating 160-162
- logging out 162, 163
- monitoring 158, 159

web server

- high availability, implementing on 87

wrapper script

- coding, for check_ora_sendtrap 266, 267

Y

yum install command 302

Z

Zabbix

- about 11, 55
- action 222
- architecture, for large environment 12, 13
- architecture, types 13
- baseline, defining 50
- capacity planning 48
- compiling, with ODBC 134
- database monitoring 48, 49, 126
- data sending, Zabbix sender
 - protocol used 276
- data, visualizing through maps 182-185
- data, visualizing through
 - slide show 201, 202
- external checks 254
- features 11
- graphs 171
- historical data, storing 53
- installing 14, 15
- installing, prerequisites 16
- IT services 205
- load testing 51, 52
- macros, using 236-241
- observer effect 48
- performance tuning 48
- RT, setting up 309

- screens, visualizing through 197
- security concerns 73, 74
- setting up, to integrate with RT 318-321
- templates, creating 234
- trigger dependencies, managing 221
- trigger expression 212
- URL, for documentation 285
- user parameter 261
- Zabbix agent protocol 268**
 - about 273-275
 - developing, considerations 280
- Zabbix agents**
 - about 13
 - setting up 18
- Zabbix API**
 - about 284, 285
 - array datatype 290
 - authenticating 286-288
 - bool datatype 289
 - data, exporting 298
 - datatypes 289, 291
 - flag datatype 290
 - graphs, creating from data 302
 - hosts, redistributing to proxy 294, 295
 - integer datatype 290
 - object datatype 291
 - operations, performing 294
 - PyZabbix library, using 288-291
 - query datatype 291
 - string datatype 290
 - tabular data, extracting 299, 301
 - timestamp datatype 290
 - trigger dependency graph, creating 303-306
 - URL 284
 - users, adding 295-297
 - users, updating 295-298
- Zabbix architecture**
 - database, considerations 30-32
 - database, installing 21-23
 - database, sizing 32-35
 - housekeeping, considerations 35-41
 - web interface, installing 41
 - Zabbix installation 14, 15
 - Zabbix server, setting up 17, 18
 - Zabbix server, starting up 24-30
- Zabbix forum**
 - URL 256
- zabbix_get command 17**
- Zabbix get protocol 268 269**
- Zabbix item**
 - aggregated items 164-166
 - calculated items 164, 167
 - data flow 122-124
 - gathering, as raw data 120, 121
 - types 125, 126
 - Zabbix trapper item 125
- Zabbix Java gateway**
 - about 137
 - installing 139-141
- Zabbix Map**
 - creating 186-188
- Zabbix node**
 - about 66
 - data flow, monitoring 67
 - deploying 67-71
 - disadvantages 72
 - vs, Zabbix proxy 72, 73
- Zabbix protocol**
 - about 268, 269
 - possible responses 275
 - Zabbix agent 268-275
 - Zabbix get 268, 269
 - Zabbix sender 268-271
- Zabbix proxy**
 - about 13, 56, 57
 - active proxy 63-65
 - architecture 56, 57
 - data flow, monitoring with 63-65
 - deploying 57-62
 - passive proxy 63-65
 - vs, Zabbix node 72, 73
- zabbix_send command 17**
- Zabbix sender protocol**
 - about 268-271
 - clock property, using 272
 - implementing, in Java 276, 277
 - implementing, in Python 278, 279
 - undocumented feature 271, 272
 - used, for sending data to Zabbix 276

zabbix_sender utility

used, for sending data 264

Zabbix server

about 13

configuring 20, 21

configuring, for high availability 96, 97

data sending, new script used 265, 266

setting up 17, 18

starting up 24-30

Zabbix trapper item 125

Zabbix Version 1.8 284

Zabbix Version 2.0

about 255

enhancements 255



Thank you for buying Mastering Zabbix

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.

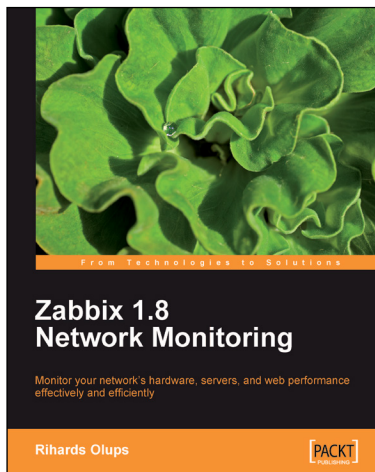
About Packt Open Source

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.



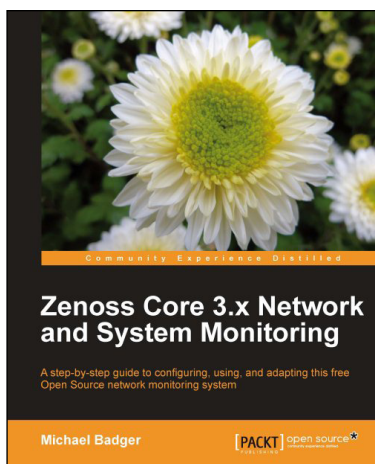
Zabbix 1.8 Network Monitoring

ISBN: 978-1-847197-68-9

Paperback: 428 pages

Monitor your network's hardware, servers, and web performance effectively and efficiently

1. Start with the very basics of Zabbix, an enterprise-class open source network monitoring solution, and move up to more advanced tasks later
2. Efficiently manage your hosts, users, and permissions
3. Get alerts and react to changes in monitored parameters by sending out e-mails, SMSs, or even execute commands on remote machines



Zenoss Core 3.x Network and System Monitoring

ISBN: 978-1-849511-58-2

Paperback: 312 pages

A step-by-step guide to configuring, using and adapting this free Open Source network monitoring system

1. Designed to quickly acquaint you with the core feature so you can customize Zenoss Core to your needs
2. Discover, manage, and monitor IT resources
3. Build custom event-processing and alerting rules
4. Write custom device reports to extract, display, and analyze monitoring data

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

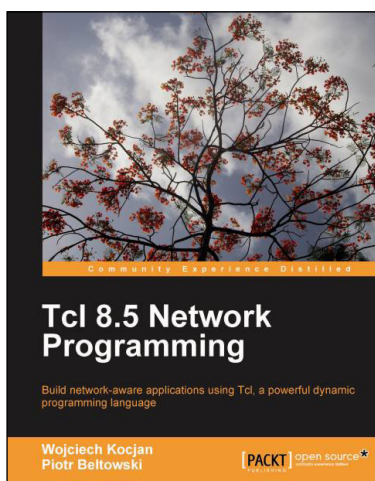


Oracle VM Manager 2.1.2

ISBN: 978-1-847197-12-2 Paperback: 244 pages

Manage a flexible and elastic data center with Oracle VM Manager

1. Learn quickly to install Oracle VM Manager and Oracle VM Servers
2. Learn to manage your Virtual Data Center using Oracle VM Manager
3. Import VMs from the Web, template, repositories, and other VM formats such as VMware
4. Learn powerful Xen Hypervisor utilities such as xm, xentop, and virsh



Tcl 8.5 Network Programming

ISBN: 978-1-849510-96-7 Paperback: 588 pages

Build network-aware applications using Tcl, a powerful dynamic programming language

1. Develop network-aware applications with Tcl
2. Implement the most important network protocols in Tcl
3. Packed with hands-on-examples, case studies, and clear explanations for better understanding

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