SWsoft

VZAgent

Programmer's Guide

1.0

(c) 1999-2007

ISBN: N/A
SWsoft
13755 Sunrise Valley Drive
Suite 325
Herndon, VA 20171
USA
Tel: +1 (703) 815 5670
Fax: +1 (703) 815 5675

© 1999-2007 SWsoft. All rights reserved.
Distribution of this work or derivative of this work in any form is prohibited unless prior written permission is obtained from the copyright holder.
Virtuozzo, Plesk, HSPcomplete, and corresponding logos are trademarks of SWsoft.
Virtuozzo is a patented virtualization technology protected by U.S. patents 7,099,948; 7,076,633; 6,961,868. Patents pending in the U.S.
Plesk and HSPcomplete are patented hosting technologies protected by U.S. patents 7,099,948; 7,076,633. Patents pending in the U.S.
Intel, Pentium, and Celeron are registered trademarks of Intel Corporation.
IBM DB2 is a registered trademark of International Business Machines Corp.
MegaRAID is a registered trademark of American Megatrends, Inc.
PowerEdge is a trademark of Dell Computer Corporation.

Contents

Preface 6

About This Guide 6
Who Should Read This Guide 6
Organization of This Guide 6
Documentation Conventions 6
- Typographical Conventions 7
- Shell Prompts in Command Examples 7
- General Conventions 8

Feedback 8

Introduction 9

VZAgent Overview 9
API 10
Installation 10
Starting, Stopping, Restarting VZAgent 11

Key Concepts 12

VZAgent Architecture 12
Connectivity 13
Authentication Concepts 14
- Realms 15
- Authorization 15

Terminology 15

Using XML API 18

XML API Basics 18
- XML Schema 18
- VZAgent Messages 19
- Error Handling 32

Creating a Simple Client Application 34
- Connecting to VZAgent 35
- Logging In 36
- Retrieving a List of Virtuozzo VEs 39
- Restarting a Virtuozzo VE 40
- Summary 41
- The Complete Program Code 42

Advanced Sample Program 44
- Encoding and Decoding an XML message 45
- The Complete Program Code 46
- Some Programming Techniques 54
- XML Libraries 55

Login and Session Management 56
- Retrieving the Realm Information 57
- Logging In 60
- Sessions 62

Creating and Configuring a Virtuozzo VE 64
- Getting a List of Sample Configurations 64

Getting a List of OS Templates 67
Populating the VE Configuration Structure....68
Creating the Virtual Environment 69
Configuring a Virtuozzo VE....70
Retrieving a VE Configuration....70
Destroying a Virtuozzo VE 70
Performance Monitoring....70
Events and Alerts....70

Using SOAP API 71

Introduction 71
Overview 71
Key Features....72
Limitations....72
The Location of WSDL 72
Generating Client Code from WSDL 72
Installation 73
Creating a Simple Client....73
Step 1: Choosing a Development Project 73
Step 2: Generating the Stubs from WSDL....73
Step 3: Adding the Code....74
Step 4: Running the Sample 77
Invoking Web Services....77
SOAP Bindings 77
Optional Elements 78
Elements with No Content....79
Base64-encoded Values....80
Polymorphism 81
Timeouts....83
Get/Set Method Name Conflict 84
Other SOAP Clients and Their Known Issues 86
Visual Basic .NET....86
Visual J# .NET 86
Apache Axis 1.2 for Java....87
Using SOAP API 87
Managing Virtual Environments 88
Performance Monitor....89
Setting Up a Cluster....94
Troubleshoting....96

Advanced Topics 97

Authentication and Authorization....97
Directory as User Database 98
Realms 99
Sessions 100
VZAgent Configuration....100
Vocabulary....100
VZAgent Group 101
Virtuozzo Group Architecture 102
Scheduler 103
Message Classification and Priorities....104
Pool and Single Operators 105
Dynamic Limits....106
Queue....106
Timeouts....107

Index 108

CHAPTER 1

Preface

In This Chapter

About This Guide 6
Who Should Read This Guide 6
Organization of This Guide 6
Documentation Conventions 6
Feedback 8

About This Guide

This guide describes how to develop VZAgent client applications. This documentation exists as XHTML pages, HTML Help (CHM), and Adobe Acrobat (PDF) documents.

Who Should Read This Guide

Primary audience for this guide is anyone developing VZAgent client applications. To use this book, you should have UNIX or Windows system administration experience and a good knowledge of Virtuozzo and its Virtual Environment management concepts. Some programming skills are required, including a good knowledge of XML and XML Schema language (also referred to as XML Schema Definition or XSD) and optionally a knowledge of SOAP and one of the languages supporting it.

Organization of This Guide

Documentation Conventions

Before you start using this guide, it is important to understand the documentation conventions used in it. For information on specialized terms used in the documentation, see the Glossary at the end of this document.

Typographical Conventions

The following kinds of formatting in the text identify special information.

Formatting convention	Type of Information	Example
Triangular Bullet(➢)	Step-by-step procedures. You can follow the instructions below to complete a specific task.	*To create a VE:*
Special Bold	Items you must select, such as menu options, command buttons, or items in a list.	Go to the **Resources** tab.
	Titles of chapters, sections, and subsections.	Read the **Basic Administration** chapter.
Italics	Used to emphasize the importance of a point, to introduce a term or to designate a command line placeholder, which is to be replaced with a real name or value.	These are the so-called *EZ templates*. To destroy a VE, type `vzctl destroy` *`veid`*.
`Monospace`	The names of commands, files, and directories.	Use `vzctl start` to start a VE.
`Preformatted`	On-screen computer output in your command-line sessions; source code in XML, C++, or other programming languages.	`Saved parameters for VE 101`
`Monospace Bold`	What you type, contrasted with on-screen computer output.	**`# rpm -V virtuozzo-release`**
CAPITALS	Names of keys on the keyboard.	SHIFT, CTRL, ALT
KEY+KEY	Key combinations for which the user must press and hold down one key and then press another.	CTRL+P, ALT+F4

Shell Prompts in Command Examples

Command line examples throughout this guide presume that you are using the Bourne-again shell (bash). Whenever a command can be run as a regular user, we will display it with a dollar sign prompt. When a command is meant to be run as root, we will display it with a hash mark prompt:

Bourne-again shell prompt	`$`
Bourne-again shell root prompt	`#`

General Conventions

Be aware of the following conventions used in this book.

- Chapters in this guide are divided into sections, which, in turn, are subdivided into subsections. For example, Documentation Conventions is a section, and General Conventions is a subsection.
- When following steps or using examples, be sure to type double-quotes ("), left single-quotes (`), and right single-quotes (') exactly as shown.
- The key referred to as RETURN is labeled ENTER on some keyboards.

The root path usually includes the `/bin`, `/sbin`, `/usr/bin` and `/usr/sbin` directories, so the steps in this book show the commands in these directories without absolute path names. Steps that use commands in other, less common, directories show the absolute paths in the examples.

Feedback

If you spot a typo in this guide, or if you have thought of a way to make this guide better, we would love to hear from you!

If you have a suggestion for improving the documentation (or any other relevant comments), try to be as specific as possible when formulating it. If you have found an error, please include the chapter/section/subsection name and some of the surrounding text so we can find it easily.

Please submit a report by e-mail to userdocs@swsoft.com.

CHAPTER 2

Introduction

In This Chapter

VZAgent Overview 9
API 10
Installation 10
Starting, Stopping, Restarting VZAgent 11

VZAgent Overview

VZAgent is a server-side software that allows client applications to connect to an manage Virtuozzo systems over network. VZagent can be used for managing, monitoring, and tuning the host server (Hardware Node) and the Virtuozzo VEs running on it. It was created in order to unify the way client applications access the Hardware Node, and to provide simple and standard means of using Virtuozzo services. The following list describes the most common tasks that can be performed through VZAgent:

- Create Virtuozzo Virtual Environments (VEs).
- Start, stop, migrate, clone, move, back up, destroy a VE.
- Obtain current statistical data and resource usage for a VE.
- Migrate existing servers (physical to virtual, virtual to physical, and virtual to virtual).
- Perform many other Virtuozzo-specific tasks. In short, you can do all of the usual management tasks that Virtuozzo allows you to do using the standard tools that come with it, but this time, you can do all of that programmatically through VZAgent.
- Shut down or restart a server.
- Manage the server configuration.
- Manage operating system services: start, stop, retrieve a list of services complete with states, settings, and run levels.
- Manage devices: retrieve device lists, mount, un-mount, format, etc.
- Manage files and directories: list, copy, move, remove, upload and download, change the owner and the mode, link, search, retrieve full sizes of directories.
- Manage users and groups: add new, remove, retrieve, etc.
- Retrieve the disk, network and other system information.
- Monitor the server resource consumption: disk, CPU, memory, network traffic.
- Receive notifications about critical server events -- directly or via e-mail.
- Back up and restore (local and remote backups).

API

VZAgent comes with an API that enables the development of client applications. The API supports the following protocols:

XML

XML API is a set of rules by which client can exchange information with and request actions from the server. The protocol is based on XML messages. The message formats are specified according to the XML Schema version 1.1 standard. With the XML API, you establish a connection with the server, compose an XML request in accordance with the schema, and send it to the server. The server processes the request, takes the appropriate action, and sends back an XML message containing the data.

SOAP

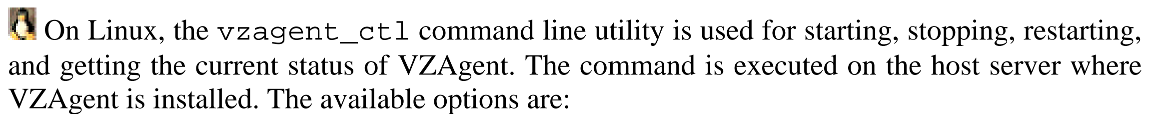
SOAP API is a standard-based Web service with language bindings for most popular languages. With SOAP API, you build your client applications using one of the third-party development tools that can generate client code from WSDL specifications. As a result, you make API calls in a language native format instead of building and parsing XML code. SOAP API shares the XML schema with the XML API.

Installation

VZAgent software is included in Virtuozzo Tools, which comes with Virtuozzo 4.0. When you install Virtuozzo on your computer, you have a choice of installing Virtuozzo Tools at that moment or at a later time. To begin using VZAgent, you have to install Virtuozzo Tools on your server.

When VZAgent is installed on your host server for the first time, you will need to know the password of your system administrator (such as `root` on Linux or `Administrator` on Windows) in order to connect to it from your client program. The system administrator is by default granted all access rights in VZAgent, which means that the user can execute any of the available VZAgent API calls and access any of the Virtuozzo VEs hosted by the Hardware Node. You can add more users with specific access rights later using Virtuozzo Tools (VZCC, VZMC) or programmatically through VZAgent.

Starting, Stopping, Restarting VZAgent


On Linux, the `vzagent_ctl` command line utility is used for starting, stopping, restarting, and getting the current status of VZAgent. The command is executed on the host server where VZAgent is installed. The available options are:

```
vzagent_ctl start
vzagent_ctl stop
vzagent_ctl restart
vzagent_ctl status
```

In the following example, the `vzagent_ctl status` command reports that VZAgent is functioning properly:

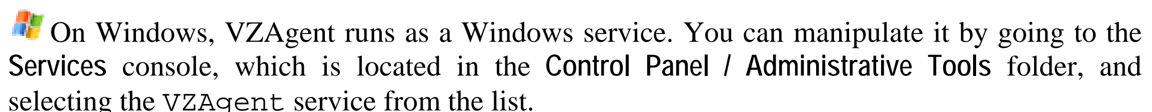
```
[root@dhcp0-190 ~]# vzagent_ctl status
vzagent (pid 31615 29644 25012 22861 8362 7073 7046 7036 7035 7029
7028 7026 7025 7023 7021 7019 7018 7017 7016 7013 7012 7011 7010 7009
7008 7007 7006 7004 7003 7002 7001 7000 6999 6998 6997 6996 6995 6994
6993 6992 6991 6990 6989 6988 6987 6986 6985 6984 6632) is running...
[root@dhcp0-190 ~]#
```

When VZAgent is stopped, the output of the same command will be as follows:

```
[root@dhcp0-190 ~]# vzagent_ctl status
vzagent is stopped
```

If something is wrong with VZAgent, the output may contain additional messages describing the problem. In such a case, try restarting VZAgent using the `vzagent_ctl restart` command:

```
[root@dhcp0-190 ~]# vzagent_ctl restart
Shutting vzagent:                          [  OK  ]
vzaproxy: no process killed
Stopping slapd: [  OK  ]
Checking configuration files for slapd:   [  OK  ]
Starting slapd: [  OK  ]
Starting vzagent:                          [  OK  ]
[root@dhcp0-190 ~]#
```


On Windows, VZAgent runs as a Windows service. You can manipulate it by going to the **Services** console, which is located in the **Control Panel / Administrative Tools** folder, and selecting the `VZAgent` service from the list.

Key Concepts

VZAgent Architecture

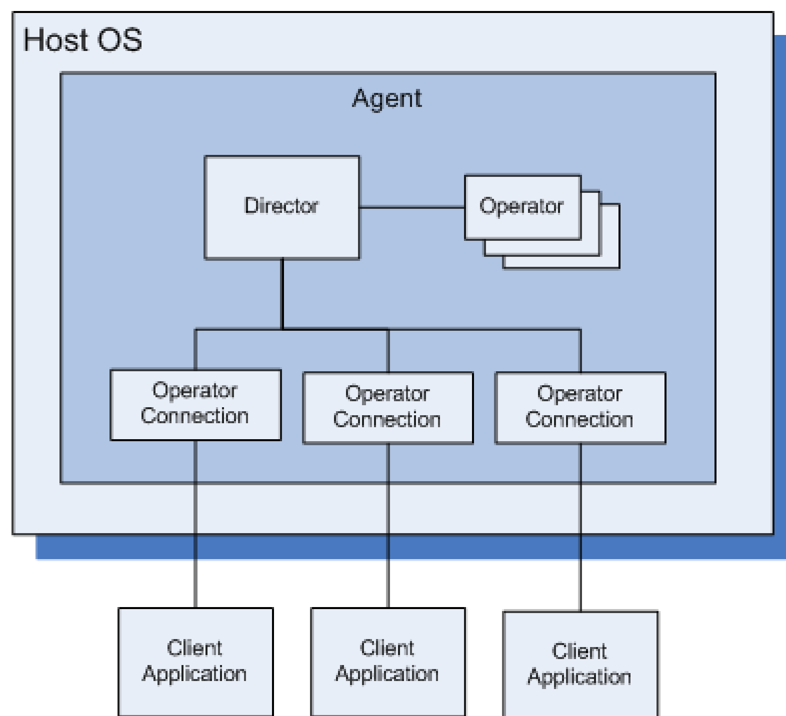


Figure 1: Agent Architecture

VZAgent is not a single executable or a single process. It is a combination of processes, communicating with each other by means of sockets or pipes. The core entities of VZAgent architecture are *operators* and *directors*.

A *director* is responsible for message routing inside VZAgent. A director determines which internal VZAgent component should serve an incoming request and to which client a particular reply should be sent.

An *operator* is a process forked or spawned from a director process. There's a collection of VZAgent operators, each of which provides a specific type of functionality. For example, the `vzaenvm` operator provides functionality for managing Virtuozzo VEs; the `vzarelocator` operator provides functionality for VE migration, etc. The diagram above illustrates the VZAgent component structure and interaction. A client connects to VZAgent through the operator *Connection* (a special operator, which is created for every client connection and which serves as a gateway between a client and a director). The client sends XML messages to the director. Based on the information provided in the request, the director determines the operator that a particular message should be sent to for processing. The operator processes the message, takes the appropriate actions, and generates a reply, which is then routed back to the client.

The operators are be divided into four major groups:

On-demand Operators are handling requests that expect an immediate action to be taken and the results to be returned to the client as soon as they are ready. An operator of this type is invoked by the director exactly once per request. Once the operator is invoked, it processes the request, takes the appropriate action, and sends the results back to the client. The on-demand operators, therefore, are used for synchronous messaging ("one request, one reply"). The majority of the VZAgent API calls are processed by the on-demand operators.

Periodic Operators (collectors) are collecting statistical data on a periodic basis. A client program may request to receive the collected data at the specified time intervals. These operators, therefore, are used for asynchronous messaging ("one request, many replies" paradigm).

Event Reporters monitor the system for critical system events, such as server configuration changes, server status changes, etc. These operators are subscription-based, meaning that the client subscribes to the event notification service and the operator notifies the client (directly or via e-mail) every time the event takes place. The client can cancel the subscription at any time.

The *System Operator* is a special operator. It controls VZAgent itself, its services, and provides some essential system functions. The operator is used to log on to VZAgent, manage VZAgent configuration, see the state of the operators and directors, verify the VZAgent version, retrieve vocabulary, subscribe to an event notification service, and to perform other system tasks.

VZAgent XML API consists of interfaces that provide access to their respective server-side operators. There's one API interface per each VZAgent operator. The name of an operator and the name of a corresponding API interface are always the same. For the complete list of interfaces, see **VZAgent XML Programmer's Reference**, which is a companion guide to this document.

Connectivity

The following table describes the connection types and protocols that can be used to communicate with VZAgent.

Connection	Description
SSL over TCP/IP	This is the recommended option for permanent connections. VZAgent is listening on port `4434` for incoming SSL connections.
TCP/IP	Plain TCP/IP connection. No encryption is used so this connection should be used with care. VZAgent is listening on port `4433` for incoming TCP/IP connections.
Unix Domain sockets	Unix-type connectivity. No encryption is used with this connection type.
Named Pipes	Windows Named Pipes. No encryption.
SOAP over HTTPS	Web Services clients.

SSH (deprecated)	This connection type is retained for compatibility purposes only and is not officially supported. The problem with SSH is that it authenticates incoming connection using its own user database, which makes it incompatible with Authentication Engine used by VZAgent.

Authentication Concepts

Before a user can send any requests to VZAgent, he/she must log on using a valid user name and password. VZAgent uses this information to verify that the user exists in the user database and that the supplied password is valid. If the user is in fact who he or she claims to be, the user security settings are retrieved from the database and the values stored in it are used to determine the user access rights.

A user database is called an *authentication database* in VZAgent. The following describes the types of the databases currently supported.

System authentication database

This is the user registry of the host operating system. This basically means that you can log on to VZagent using an account that exists in the operating system of the host server. In fact, when VZAgent is first installed, the only account that you can use to log on to it is the system administrator account, such as the `root` user in Linux or the `Administrator` user in Windows. By default, the host system administrator is granted all access rights in VZAgent, meaning that the user can execute any of the VZAgent API calls, and that the user has full access to the Hardware Node and all of its Virtuozzo VEs.

Internal authentication database

VZAgent comes with it's own internal authentication database. This database is used to store the Virtuozzo and VZAgent specific authentication information. For example, the built-in security roles of Virtuozzo Tools (VZCC, VZMC) are stored in this database. You can use this database to store your own VZAgent users. In addition, the database is used to store the VZAgent-specific security profiles (permissions and access rights) for the users stored in the System authentication database and for the external users (described below).

External authentication database (LDAP-compliant directory)

The third authentication database type supported by VZAgent is an external LDAP-compliant directory, such as Active Directory or ADAM on Windows, or OpenLDAP on Linux. VZAgent can authenticate users against an existing directory that is already populated with users and groups. This gives you a flexible way of using your existing user databases without duplicating the users in the VZAgent internal database. The only thing that you will have to do is to create the VZAgent security profiles for these users. This can be done through Virtuozzo Tools, such as VZCC or VZMC, or programmatically through VZAgent. The security profiles will be stored in the VZAgent internal database and will be internally linked to the profiles stored in the external LDAP directory.

Realms

VZAgent API uses the term *realm* to represent an authentication database. In VZAgent API context, a realm is the authentication database definition consisting of the parameters that describe the type of the database, its name, the connection parameters, and the database ID which in VZAgent is called *realm ID*. The realm ID is generated automatically by VZAgent at the time the realm definition is created. Realm definitions are stored in the VZAgent configuration on the host server. Before you can use an authentication database, it must be defined as a realm in the VZAgent configuration. At least two realms are defined at the time VZAgent is installed: the System realm and the Internal realm. If you are planning on using an external LDAP directory, you will have to create a realm representing it.

Authorization

The authorization in VZAgent is based on the concept of security roles. A *security role* is identified by its unique name and contains a list of VZAgent tasks that it is allowed to perform. An administrator would first create a security role granting the necessary VZAgent access rights to it. An administrator would then create a role assignment. *Role assignment* is a logical grouping of users belonging to the same security role. Role assignment has one more property called scope. *Scope* is the logical area of the Virtuozzo system where this role assignment is allowed to operate, such as the entire Hardware Node or just certain Virtuozzo VEs (other scope types exists and will be discussed later).

For example, you can create a security role that can start, stop, and restart a Virtuozzo VE. You can then create a user (or multiple users) and add them to that role. At the same time, you create a scope containing a list of some existing Virtuozzo VEs and select it to be the scope of that role assignment. As a result, your user(s) will be allowed to start, stop, and restart the VEs specified in the scope. They will not be allowed to perform any other operation (such as destroying a VE for example), and they will not have access to other VEs that may exist on the same host.

Terminology

This section describes some of the VZAgent terminology. Please take a moment to familiarize yourself with it so that you can use the documentation efficiently. The table below describes the main terms and provides examples of how a term is related to the various aspects of the system.

Term	Description
Virtualization Technology	Any server virtualization product that is supported by and can be accessed through VZAgent. **Note:** At the time of this writing, the only supported virtualization technology is Virtuozzo.

Environment	In a network of interconnected computer systems utilizing server virtualization technologies, each node may represent a physical or a virtual server. We use the term *Environment* to identify any computer system, whether physical or virtual, within such a network. Virtuozzo uses the term *Virtual Environment*, or *VE*, for its virtual servers.
Environment ID (EID)	Every computer within a VZAgent infrastructure is automatically assigned a universally unique ID. Once VZAgent is installed on a server, the server itself is assigned an EID, and every Virtuozzo VE that you create on it, will also be assigned an EID. All operations on Environments are performed by referencing them through the Environment ID.
Virtual Environment ID (VEID)	A VEID is a Virtuozzo-level, user-assigned Virtual Environment ID. The ID is unique only within a context of a given Hardware Node. This ID is not to be confused with the EID described above, which is a universally unique VZAgent-level Environment ID. Most VZAgent calls use EID when referencing an Environment. Some Virtuozzo-specific calls also use VEID as a parameter.
Virtuozzo group Master Environment Slave Environment	The term *Virtuozzo group* refers to a network of Environments each running its own VZAgent software and interconnected with each other by means of internal VZAgent mechanisms. The Environments in a group are organized in a hierarchical structure where there's one *Master Environment* and many *Slave Environments*. The master Environment administers the entire group by allocating, monitoring, and controlling the group resources. Master is also capable of accessing any slave Environment in a group, meaning that once a client program is connected to master, it can send requests to any of the slave Environments in the group. We will talk about Virtuozzo group in more detail later in this guide.
Parent Environment	The term *parent Environment* refers to a server (physical or virtual) that hosts other virtual servers. A server, therefore, is said to be a parent Environment to the virtual Environments that it hosts. Note that virtual Environments may host other virtual Environments -- theoretically speaking, the number of levels of the hierarchy is not limited. If the Environments are also interconnected into a Virtuozzo group, the hierarchy becomes even more complex. In such hierarchy, each virtual Environment has a parent Environment. The term *parent Environment* is not to be confused with the term *master* E*nvironment* described above. When we say *master*, we refer to the master of the entire Virtuozzo group. Master Environment can also be a parent Environment if it hosts virtual servers.

Root Environment	In general, describes an Environment that is an entry point in the Environment hierarchy. In this documentation, we use the term to identify the Environment that the client program is currently connected to. It could be any type of Environment -- virtual or physical -- and can be located anywhere in the Virtuozzo group hierarchy. VZAgent provides methods for a direct access to the root Environment and all its child Environments, i.e. the virtual Environments hosted by the root Environment.
Realm	A realm is an authentication database containing the VZAgent user and group data. VZAgent supports a number of different "databases" including operating system user registries and LDAP-compliant directories. Realm definitions are stored in the VZAgent configuration. Every realm is assigned a universally unique ID by VZAgent.

CHAPTER 3

Using XML API

The material in this chapter is intended for the developers who would like to develop VZAgent client applications using XML API. This chapter does not provide general information on XML. We assume that you are comfortable working with XML and have some experience working with XML Schema language (also referred to as XML Schema Definition or XSD).

In This Chapter

XML API Basics 18
Creating a Simple Client Application 34
Advanced Sample Program 44
Login and Session Management 56
Creating and Configuring a Virtuozzo VE 64
Performance Monitoring 70
Events and Alerts 70

XML API Basics

XML API provides programmatic access to VZAgent. It can be described as a set of rules by which client can exchange information with and request actions from VZAgent. This chapter describes the main principles of the XML API and technologies it is built upon. It then provides technical details on the XML message structure complete with guidelines and examples. It concludes with the description of error handling in API calls.

XML Schema

The XML Schema is a document that represents how the XML data must be organized. It defines the data elements that are required and those that are optional. The schema also describes the data types, and the relationship between the data elements (single or multiple instance, parent and child elements).

You make an XML API call by first establishing a connection with VZAgent, then composing an XML message according to the schema specifications, and finally sending the message to VZAgent. You similarly receive responses from VZAgent as XML messages. The XML API schema defines every single message that you can send and receive. This means that every possible request and response message is strictly defined and must be structured and formatted according to schema specifications. When composing a request, the schema must be strictly followed in order for the message to be understood by VZAgent. Failure to do so will result in error.

VZAgent XML Programmer's Reference contains the complete XML schema with descriptions, guidelines, and code examples.

VZAgent Messages

In order to build VZAgent XML messages correctly and to take full advantage of the available options, it is important to understand the basic building blocks of a message. This section focuses on how a VZAgent message is organized, and provides the necessary specifications and examples.

A VZAgent message is an XML document composed of elements. The elements may contain other elements or they may contain data. Elements may also contain attributes.

The two main components of a VZAgent XML message is the *header* and the *body*. The header provides the packet routing and control information. The body of the packet contains the actual request (or the response) message. The following table contains an examples of a valid VZAgent request message.

XML message (request)	Description
<packet version="4.0.0" id="2">	This is the root element of any message. The id attribute specifies the packet ID. The version attribute specifies the protocol version.
<target>sessionm</target>	The target VZAgent operator that the request should be sent to for processing. **Note:** When using the system operator, do not include the target element. The system operator is the only exception. All other operators require the target element.
<data>	The data block contains the message body.
<sessionm>	Every request begins with the name of the interface providing the desired functionality. The interface name is always the same as the name of the operator (see target element above).
<login>	This is the name of the API call.
<name>bXluYW1l</name>	This and the following elements are the API call parameters.
<realm>00000000</realm>	Parameter.
<password>bXlwYXNz</password>	Parameter.
</login>	Closing tag.
</sessionm>	Closing tag .
</data>	Closing tag .
</packet>	Closing tag .

A response message may look similar to the following example.

XML Message (response)	Description
<packet id="2" time="2007-03-11T11:17:30+0000" priority="0" version="4.0.0">	The root element. The time attribute specifies the response date and time. The version attribute specifies the protocol version.
<origin>sessionm</origin>	The name of the operator that processed the request and generated this response as a result.
<target>vzclient1</target>	The client who sent the initial request message. This value is generated and used internally by VZAgent.
<data>	The message body.
<sessionm>	Just like a request message, every response message also begins with the name of the interface. The block that follows this element contains the returned data.
<token>	Data.
<user>AQUAAAAAIAFWKop...</user>	Data.
<groups>	Data.
<sid>AQUAAAAAIABWKop..</sid>	Data.
</groups>	Closing tag.
</token>	Closing tag.
</sessionm>	Closing tag.
</data>	Closing tag.
</packet>	Closing tag.

Schema Tables

The XML schemas in VZAgent documentation are presented as tables where each row represents an XML element. The following is an example of a table containing an XML API call specification:

Name	Min/Max	Type	Description
login			
{			
name	0..1	base64Binary	User name.
domain	0..1	base64Binary	Domain.
realm	1..1	guid_type	Realm ID.
password	0..1	base64Binary	User password.
expiration	0..1	int	Custom timeout value.

}			

The following describes the table columns:

Name

Specifies the XML element name. The curly brackets represent the XML Schema sequence element (`<xs:sequence>`). This means that the elements inside the brackets are the child elements of the element that precedes the opening bracket. In our example, the `name`, `domain`, `realm`, `password`, and `expiration` elements are children of the `login` element. Therefore, the XML document composed using this specification would look like this:

```
<login>
  <name>user_name</name>
  <domain>domain_name</domain>
  <realm>realm_id</realm>
  <password>user_password</password>
  <expiration>exp_value</expiration>
</login>
```

Min/Max

Specifies the cardinality of an element (the number of its minimum and maximum occurrences) in the following format:

`minOccurs..maxOccurs`

`0` in the first position indicates that the element is optional.

`1` in the first position indicates that the element is mandatory and must occur at least once.

A number in the second position indicates the maximum number of occurrences. The `[]` (square brackets) in the second position indicate that the number of the element occurrences is unbounded, meaning that the element may occur as many times as necessary in the same XML document at the specified position.

The following examples demonstrate how an element cardinality may be specified:

`0..1` The element is optional and may occur only once if included in the document.

`1..1` The element is mandatory. One, and only one, occurrence is expected in the document.

`0..[]` The element is optional but may occur an unlimited number of times if needed.

`1..[]` The element is mandatory. At least one occurrence is expected but the element may occur an unlimited number of times if needed.

Type

Specifies the element type. The following element types are used in the schema:

- Standard simple types: `int, string, base64Binary,` etc.
- Custom simple types. These types are usually derived from standard simple types with additional restrictions imposed on them.
- Custom complex types.

Description

The description column contains the element description and provides information on how to use it.

Message Header

Summary:

The packet element is the root element of every message. Both the header and the body of a message reside within the same parent packet element. The following XML Schema specification defines the VZAgent message header.

Message header specification:

Name	Min..Max	Type	Description
packet			The root element of any VZAgent message.
{			
cookie	0..1	string	User-defined information describing the packet. The data specified here remains unchanged during the request/response operation.
target	0..[]	string	In request messages, this element must contain the name of the operator to which to send the request for processing. **Note:** When using the system operator, do not include the target element. The system operator is the only exception. All other operators require the target. The name of the operator here is always the same as the name of the corresponding interface that you are using. For example, if you are using a call from the vzaenvm interface, the name of the target operator is also vzaenvm. Multiple targets may be specified if you are including multiple calls in a single request. In response messages, this element contains the name of the client that originated the request (the value is generated and used internally by VZAgent).
origin	0..1	string	The name of the operator that generated the response. Included in response messages only.

src	0..1	routeType	The source routing information. This field is automatically populated by the director on the server side when a message is routed from the corresponding operator to it. The same information is also duplicated in the dst element (described below) when a response is generated and is sent back to the client.
{			
director	0..1	string	The name of the director to which the target operator belongs.
host	0..1	string	The VZAgent host ID. Used by VZAgent to determine the host address. Should be either contained in VZAgent configuration (global mapping) or be a result of exclusive connect.
index	0..1	string	For on-demand operators specifies a particular target.
target	0..1	string	Contains the origin information when a packet is sent remotely.
}			
dst		routeType	The destination routing information. In the request messages, use this field to specify the Environment to which you would like to forward the request. For example, if you are sending a request to the VZAgent on the Hardware Node but would like the request to be processed inside a Virtuozzo VE, specify the EID fo the VE using the dst/host parameter. In the response messages, this information is automatically populated by the director on the server side.
{			
director	0..1	string	The name of the director to which the target operator belongs. Populated automatically by the director.
host	0..1	string	The VZAgent host ID. When using the message forwarding feature, used for specifying the EID of the target Environment.
index	0..1	string	For on-demand operators specifies a particular target. Populated automatically by the director.
target	0..1	string	Contains the origin information when a packet is sent remotely. Populated automatically by VZAgent.
}			

session		string	The session ID. In the request messages, this field is used to specify the session that should be used to process the request. In the response messages, the ID indicates the session that was used to process the request. The session ID is obtained from the response message of the sessionm/login API call after a successful login.
}			

The packet element may optionally contain the attributes described in the following table.

Attributes of the <packet> element:

Attribute	Type	Description
version	string	The VZAgent protocol version number. The current protocol version number is 4.0.0. The older 3.0.3 protocol is also supported in Virtuozzo 4.0.
id	string	Packet ID. If included in a request message, the response will contain the same ID value. This allows to easily match requests and responses in multi-request/response situations. The attribute must also be included if you want to be notified in case of the request timeout, or if the packet was dropped on the server site for some reason. As a rule of thumb, you should always include the ID in all of your outgoing packets. The value should normally be a string containing an integer value, but it can also contain other characters if needed.
priority	string	Packet priority. Specifies the significance of the message when it is placed into a message queue. The higher the priority value, the less significant the packet is. The zero value is the default priority. Priorities range from -3000 to 3000. -3000 to -1000 for heavy messages. -999 to 999 for normal messages. 1000 to 3000 for urgent messages.
time	datetime_type	The time when the packet was sent; in the ISO-8601 format: (e.g. "2007-02-04T08:55:51+0000").

`progress`	`string`	Use this attribute to enable the progress reporting for long operations if you would like to receive intermediate results and to keep track of the request processing. Please note that not all operations actually generate progress reports. The possible values are: `on` -- the progress reporting is on. `off` (default if the attribute is omitted) -- the progress reporting is off. When you turn the progress reporting on, you must also include the `id` attribute (above) specifying the message ID.
`log`	`string`	When present, the automatic progress reporting is logged for the operations supporting it. Switch this to "on" if you're planning to start an operation and disconnect from VZAgent before the operation is completed. By doing so, you'll be able to reconnect later and check the log files for the results of your operation. The requests marked as *Logged Operation* in the XML API Reference support this feature. Possible values are: `on` -- the logging is turned on. `off` (default) -- the logging is off.
`type`	`int`	*** INTERNAL *** Bit field for the internal type of the message. #define UNFINISHED 0x00000001 #define RESPONSE 0x00000002 #define RESCHEDULE 0x00000004 #define TIMEOUT 0x00000008
`timeout`	`int`	The timeout value which will be used for handling this request. The value can be specified in the incoming packet or it can be sent back from the operator, notifying the director about the time it is going to handle it.
`timeout_limit`	`int`	*** INTERNAL *** Timeout limit for message processing. Used by an operator in determining the validity of its timeout.
`uid`	`int`	*** INTERNAL *** UID of the user sending this packet.

Example:

The following is an example of a VZAgent message header built according to the specifications above. In a real message, the values of the XML elements would be substituted with the appropriate names, IDs, etc.

```
<packet version="4.0.0" id="500">
  <cookie>I'm a cookie holding some text</cookie>
  <target>operator_name</target>
  <dst>
    <host>target_Environment_ID</host>
  </dst>
  <session>session_id</session>
</packet>
```

Message Body

Message body contains the actual request/response message. The `data` element is the root element of the message body tree. It is followed by the name of the interface that you would like to use, the name of the call, and the call parameters.

Note: There must be one and only one `data` element in any given message.

The request message:

The following XML code example is a complete VZAgent request message. As you already know, the `packet` element is the root element of every VZAgent message. The `target` element specifies the name of the target operator. The message body begins with the `data` element. The `sessionm` element specifies the name of the interface. The available interfaces are documented in the **VZAgent XML Programmer's Reference** documentation. The `login` element is the name of the call. The `name`, `realm`, and `password` elements are the call parameters.

```
<packet version="4.0.0" id="2">
  <target>sessionm</target>
  <data>
    <sessionm>
      <login>
        <name>cm9vdA==</name>
        <realm>00000000-0000-0000-0000-000000000000</realm>
        <password>bXlwYXNz</password>
      </login>
    </sessionm>
  </data>
</packet>
```

The response message:

The following example demonstrates a complete response message. The body of the message begins with the `<data>` element which is followed by the name of the interface that was used in the corresponding request message, and the return parameters.

```
<packet xmlns:ns2="http://www.swsoft.com/webservices/vzl/4.0.0/types"
xmlns:ns1="http://www.swsoft.com/webservices/vzl/4.0.0/sessionm"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
id="8c46e52645t18berd40" time="2007-09-09T02:11:21+0000" priority="0"
version="4.0.0">
<origin>sessionm</origin>
<target>vzclient69-4fce28dd-0cd3-1345-bb94-3192b940fb90</target>
<dst>
  <director>gend</director>
</dst>
<data>
  <sessionm>
    <session_id>vzl.40000.4.4fce28dd-0cd3-13..</session_id>
    <token xsi:type="ns2:tokenType">
      <user>AQUAAAAAIAHdKM5P0wxFE7uUMZK5QPuQAAAAAA==</user>
      <groups>
        <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQAAAAAA==</sid>
        <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQAQAAAA==</sid>
        <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQCgAAAA==</sid>
        <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQAgAAAA==</sid>
```

```
        <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQAwAAAA==</sid>
        <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQBAAAAA==</sid>
        <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQBgAAAA==</sid>
        <sid>AQUAAAAAIAHdKM5P0wxFE7uUMZK5QPuQAAAAAA==</sid>
      </groups>
      <deny_only_sids/>
      <privileges/>
    </token>
  </sessionm>
</data>
<src>
  <director>gend</director>
</src>
</packet>
```

A body of the esponse message may, in general, contain one of the following types of information:

- The actual information requested, as shown in the example above.
- The `<OK/>` element if the call doesn't return any data.
- The error information, in case of a failure.

A complete XML Schema specification exists for every possible response of every VZAgent XML API call, and is described in the corresponding section of **VZAgent XML Programmer's Reference**.

Multiple Calls and Targets

You may include more than one API call in a single request message. The calls may belong to the same interface/operator or they may belong to different operators. For example, you may start a VE performance monitor and at the same time subscribe for an event notification. Or retrieve a list of disks from the Hardware Node and at the same time retrieve a list of devices from it. There are a few simple rules that you should follow when making multiple calls in the same message. If all calls belong to the same operator, simply specify the operator name in the `target` element in the message header and list the calls, one after another, in the message body. If the calls belong to different operators, include a separate `target` element containing the name of the operator for each call, and include the API calls in the message body. The calls are processed on the server side independently from each other. If one of the calls fails, the other calls will still be processed. The response messages are sent back for each call individually.

Example

The following request message contains two calls: one retrieves the information about the devices from the Hardware Node, and the other retrieves the information about the disks and partitions.

Input

```
<packet version="4.0.0" id="2">
  <target>vzadevm</target>
  <target>computerm</target>
  <data>
    <vzadevm>
      <get_info/>
    </vzadevm>
    <computerm>
      <get_disk/>
    </computerm>
  </data>
</packet>
```

Each call generates an individual response.

Output 1

```
<packet xmlns:ns2="http://www.swsoft.com/webservices/vzl/4.0.0/types"
xmlns:ns1="http://www.swsoft.com/webservices/vzl/4.0.0/computerm"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
id="bc46e53091t3d6crd40" time="2007-09-09T02:39:02+0000" priority="0"
version="4.0.0">
<origin>computerm</origin>
<target>vzclient69-4fce28dd-0cd3-1345-bb94-3192b940fb90</target>
<dst>
  <director>gend</director>
</dst>
<data>
  <computerm>
    <disk>
      <partition>
        <name>/dev/sda2</name>
        <mount_point>/</mount_point>
        <block_size>4096</block_size>
        <fs_type>ext3</fs_type>
```

```
        <option>rw</option>
        <blocks xsi:type="ns2:resourceType">
          <total>1239079</total>
          <used>344363</used>
          <free>830758</free>
        </blocks>
        <inodes xsi:type="ns2:resourceType">
          <total>1280000</total>
          <used>45322</used>
          <free>1234678</free>
        </inodes>
      </partition>

    <!-- The rest of the output is omitted for brevity -->

    </disk>
  </computerm>
</data>
<src>
  <director>gend</director>
</src>
</packet>
```

Output 2

```
<packet
xmlns:ns1="http://www.swsoft.com/webservices/vza/4.0.0/vzadevm"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
id="bc46e53091t3d6crd40" time="2007-09-09T02:39:03+0000" priority="0"
version="4.0.0">
<origin>vzadevm</origin>
<target>vzclient69-4fce28dd-0cd3-1345-bb94-3192b940fb90</target>
<dst>
  <director>gend</director>
</dst>
<data>
  <vzadevm>
    <device_info>
      <partition>/dev/sda1</partition>
      <partition>/dev/sda2</partition>
      <partition>/dev/sda3</partition>
      <partition>/dev/sda5</partition>
      <partition>/dev/dm-0</partition>
      <filesystem>auto</filesystem>
    </device_info>
  </vzadevm>
</data>
<src>
  <director>gend</director>
</src>
</packet>
```

The Null-Terminating Character

When an XML request message is sent to VZAgent from a client program, it must be terminated with a binary zero character (written as `'\0'`). The null-terminating character is used by VZAgent to determine the end of the message.

Namespaces

Error Handling

When an error occurs during the request processing, the error information is returned to the client as an XML message. A single response message may contain multiple errors if the original request contained more than one request. A single request may also produce more than one error message. The error information is included in the message body and may be placed at the various levels of the message body hierarchy depending on the original location of the request or the element that caused the error. The format of the XML structure containing the error information is as follows:

```
<data>
  <operator_name>
    <error>
      <code>error_code</code>
      <message>error_message</message>
    </error>
  </operator_name>
</data>
```

The element that we described as *operator_name* in the example above will actually have the same name as the VZAgent operator that generated the response. The error information consists of a numeric code and a string describing the problem. VZAgent has its own list of errors. The errors reported by various system utilities and the internal calls invoked by VZAgent operators are automatically translated to their client-level VZAgent equivalents. This means that regardless of the computing platform, the error codes and descriptions will always be consistent.

The following is an example of an error message produced by the `login` call of the `sessionm` interface.

Input:

```
<packet version="4.0.0" id="2">
  <target>sessionm</target>
  <data>
    <sessionm>
      <login>
        <name>cm9vdA==</name>
        <realm>00000000-0000-0000-0000-000000000000</realm>
        <password>bXlwYXNz</password>
      </login>
    </sessionm>
  </data>
</packet>
```

Output:

```
<?xml version="1.0" encoding="UTF-8"?><packet
xmlns:ns1="http://www.swsoft.com/webservices/vzl/4.0.0/sessionm"
xmlns:ns2="http://www.swsoft.com/webservices/vzl/4.0.0/protocol"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
id="8c46e129f3t18ber330" time="2007-09-07T05:04:50+0000" priority="0"
version="4.0.0">
<origin>sessionm</origin>
<target>vzclient67-4fce28dd-0cd3-1345-bb94-3192b940fb90</target>
<dst>
  <director>gend</director>
```

```
</dst>
<data>
  <sessionm>
    <error>
      <code>400</code>
      <message>Invalid packet: invalid password.</message>
    </error>
  </sessionm>
</data>
<src>
  <director>gend</director>
</src>
</packet>
```

Creating a Simple Client Application

In this section, we'll create a simple client program that will connect to VZAgent, submit an XML request to it, and will receive a response. We will be using Perl to write our sample program. The complete program code is included in **The Complete Program Code** section (see page 42).

If you are using Linux, you probably have Perl already installed on your machine. If you are using Windows, you can download Perl for Windows from the Internet. We will be communicating with VZAgent using SSL over TCP/IP transport. This will require a special package providing SSL support for Perl installed on your machine. Assuming that you have Perl installed and working properly, let's add the SSL support.

In our example we use the `IO::Socket::SSL` package, which can be downloaded from CPAN here: http://search.cpan.org/~behroozi/IO-Socket-SSL-0.97/SSL.pm.

The package requires another module called `Net::SSLeay`, which also can be downloaded from CPAN by going to this URL: http://search.cpan.org/~flora/Net_SSLeay.pm-1.30/SSLeay.pm.

Both modules come with extensive documentation and easy-to-follow installation instructions.

Alternatively, you can use the plain TCP/IP transport for the purpose of this simple test. Simply substitute all occurrences of `SSL` in the code with `INET.` Also, use port `4433` in the function that creates a socket (see the next section). If you decide to use TCP/IP, you don't have to install the SSL packages.

Now that we have our development environment set up, we are ready to write our program. Please note that our program will be as basic as it can possibly be. It should suffice, however, as an entry point into VZAgent programming.

Create a new text file named `agent_sample.pl` and paste or type the following code into it:

```
#!/usr/bin/perl -w

use strict;
use IO::Socket::SSL;
```

In the code above, we instruct Perl to use the `IO::Socket::SSL` package (to communicate with VZAgent using SSL over TCP/IP).

We'll now add some constants to our code:

```
#Message terminating character.
use constant MSG_TERMINATOR => "\0";
$/ = &MSG_TERMINATOR;
```

The constant is a binary zero character that we'll be appending to every message that we send (every message sent to VZAgent must be null-terminated).

Connecting to VZAgent

Let's now add the code that will establish a secure connection with VZAgent.

```
#Socket variable
my $socket;

#Create the socket
$socket = new IO::Socket::SSL( PeerAddr => '192.168.0.116', PeerPort
=> '4434', Proto => 'tcp' );

#Read the output and display it on the screen.
my $buf = <$socket>;
chomp( $buf );
print $buf;
```

In the code above, we declare a variable for our SSL socket and then create a socket by executing the `new IO::Socket::SSL()` directive. We pass the IP address of the computer where we have our VZAgent installed (you must substitute it with the actual address of your server) and the port number (`4434` is the standard port that VZAgent is listening at for SSL connections). We then read the output from a socket into a buffer and display the contents of the buffer on the screen.

Save the file now and run the program by typing `perl agent_sample.pl` at the command prompt. You should see the output similar to the following example:

```
<packet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" id="0"
priority="0" version="4.0.0">
  <origin>vzclient6-5850cc75-cbde-784c-be21-026fcd46c9d7</origin>
  <target>agent</target>
  <data>
    <ok/>
    <eid>5850cc75-cbde-784c-be21-026fcd46c9d7</eid>
  </data>
</packet>
```

If you see the above message on the screen, then it means that VZAgent is functioning properly, and that you can connect to it. If you don't see the message, make sure that VZAgent is running (see page 11) and that you can ping the server from your client computer. If you are receiving an error message, and it is related to SSL (i.e. you see some mentioning of SSL or Socket:SSL), make sure that the `IO::Socket::SSL` package is installed correctly and that you are using the correct version of Perl in accordance with the `IO::Socket::SSL` requirements.

Let's now examine the response that we received from VZAgent. As you can see, it is an XML document. In fact, this is the very first message that you receive from VZAgent every time you connect to it. It is basically a greeting message from VZAgent, which means that the initial connection has been established successfully. The `eid` element contains the Environment ID (see page 15) of the Hardware Node that your program is now connected to.

Logging In

Once you are connected to VZAgent, the first thing that you have to do is log in. You do that by executing the `login` API call from the `system` interface supplying the user credentials, including the user name, password, and the realm ID (see page 15). Since you may not know the realm ID in advance, you would normally retrieve the list of realms using the `system.get_realm` call. This is the only call that does not require you to be logged on in order to execute it. First, we have to compose our message:

```
#XML message. Retrieving the list of realms from the Hardware Node.
my $request=qq~
<packet version="4.0.0" id="2">
  <data>
    <system>
      <get_realm/>
    </system>
  </data>
</packet>
~;
```

The `$request` variable in the code above now contains our XML message. The next code segment will write our XML message to the socket that we created earlier:

```
#Write the XML message to the socket.
print $socket $request.$/;
```

Please note that we appended the binary zero character contained in the `$/` variable to the message. Failure to do so will make the request unrecognizable to VZAgent. Once again, we will read the output from the socket and will display it on the screen.

```
#Read the response and display it on the screen.
$buf = <$socket>;
chomp( $buf );
print $buf;
```

The response to this message will contain the complete list of realms defined in the VZAgent configuration on the Hardware Node. It will look similar to the following :

```
<packet id="2" version="4.0.0" priority="0">
  <origin>system</origin>
  <target>vzclient8-4fce28dd-0cd3-1345-bb94-3192b940fb90</target>
  <data>
    <system>
      <realms>
        <realm>
          <login>
            <name>Y249dnphZ2VudCxkYz1WWkw=</name>
            <realm>458d583f-f2d8-7940-a9d3-a9a3d2ec1509</realm>
          </login>
          <builtin/>
          <name>Virtuozzo Internal</name>
          <type>1</type>
          <id>458d583f-f2d8-7940-a9d3-a9a3d2ec1509</id>
          <address>vzsveaddress</address>
          <port>389</port>
          <base_dn>ou=4fce28dd-0cd3-1345-bb94-
3192b940fb90,dc=vzl</base_dn>
          <default_dn>cn=users,ou=4fce28dd-0cd3-1345-bb94-
3192b940fb90,dc=vzl</default_dn>
        </realm>
```

```
        <realm>
          <builtin/>
          <name>System</name>
          <type>0</type>
          <id>00000000-0000-0000-0000-000000000000</id>
        </realm>
        <realm>
          <builtin/>
          <name>Virtuozzo VE</name>
          <type>1000</type>
          <id>00000000-0000-0000-0100-000000000000</id>
        </realm>
      </realms>
    </system>
  </data>
</packet>
```

Assuming that Virtuozzo has just been installed on our system, we will use the system administrator account to log in to VZAgent (see **Installation** (on page 10) for more info). The System realm (see page 14) from the output above refers to the user registry on the host OS, so this is the realm that we want. In order to log in, you will also need to know the administrator password. In the following example, we are logging in to VZAgent installed on a Linux system using the `root` account. Don't forget to substitute the password value with your `root` password. If you are using a Windows-based system, use your Windows administrator account. Please note that the name, realm, and password values are Base64-encoded in accordance with the schema.

```
#XML message. Logging in.
$request=qq~
<packet version="4.0.0" id="3">
  <data>
    <system>
      <login>
        <name>cm9vdA==</name>
        <realm>00000000-0000-0000-0000-000000000000</realm>
        <password>MXEydzNl</password>
      </login>
    </system>
  </data>
</packet>
~;
```

We will now write the XML message to the socket the same way we did when we were retrieving realms in the previous step.

```
#Write the XML message to the socket.
print $socket $request.$/;
```

Once again, we are reading the output from the socket and displaying it on the screen.

```
#Read the response and display it on the screen.
$buf = <$socket>;
chomp( $buf );
print $buf;
```

If the supplied credentials were valid, the response message will contain a security token, and will look similar to the following example:

```
<packet id="3" priority="0" version="4.0.0">
  <origin>system</origin>
  <target>vzclient19-4fce28dd-0cd3-1345-bb94-3192b940fb90</target>
  <data>
```

```
    <system>
      <token>
        <user>AQUAAAAAIAHdKM5P0wxFE7uUMZK5QPuQAAAAAA==</user>
        <groups>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQAAAAAA==</sid>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQAQAAAA==</sid>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQCgAAAA==</sid>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQAgAAAA==</sid>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQAwAAAA==</sid>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQBAAAAA==</sid>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQBgAAAA==</sid>
          <sid>AQUAAAAAIAHdKM5P0wxFE7uUMZK5QPuQAAAAAA==</sid>
        </groups>
        <deny_only_sids/>
        <privileges/>
      </token>
    </system>
  </data>
</packet>
```

If you see a message like that on your screen, it means that you are now logged on to VZAgent and that a permanent session has been created for the user. A permanent session is associated with the physical connection that we've established earlier and it never expires.

Let's examine the rest of the elements in the response message. The `packet` element contains the message ID, which, as you can see, is the same ID that we specified in the request message. The `target` element contains the ID of our client connection (the value is assigned and used by VZAgent internally). The `origin` element contains the name of the VZAgent operator that processed the request on the server side. The `user` element contains the SID (security ID) of the user. The `sid` elements within the `groups` element contain the security IDs of the groups to which the user belongs as a member.

The next request that we are going to send to VZAgent will retrieve a list of Virtuozzo VEs installed on the Hardware Node.

Retrieving a List of Virtuozzo VEs

To retrieve a list of VEs installed on the Hardware Node, we will use the `get_list` API call from the `vzaenvm` interface (Virtuozzo VE management).

```
#XML message. Getting a list of Virtuozzo VEs.
$request=qq~
<packet version="4.0.0" id="4">
  <target>vzaenvm</target>
  <data>
    <vzaenvm>
      <get_list/>
    </vzaenvm>
  </data>
</packet>
~;

#Write the XML message to the socket.
print $socket $request.$/;

#Read the response and display it on the screen.
$buf = <$socket>;
chomp( $buf );
print $buf;
```

The response contains a list of Environment IDs (see page 15). The following is an example of the response message:

```
<packet id="4" time="2007-08-29T22:51:52+0000" priority="0"
version="4.0.0">
  <origin>vzaenvm</origin>
  <target>vzclient24-4fce28dd-0cd3-1345-bb94-3192b940fb90</target>
  <dst>
    <director>gend</director>
  </dst>
  <data>
    <vzaenvm>
      <eid>ba92bfb3-d97b-014f-a754-5b30528477c3</eid>
      <eid>3e25fee2-1163-4336-9e74-8b8097936d47</eid>
      <eid>72145bf0-7562-43d4-b707-cc33d37e3f10</eid>
      <eid>6dbd99dc-f212-45de-a5f4-ddb78a2b5280</eid>
    </vzaenvm>
  </data>
  <src>
    <director>gend</director>
  </src>
</packet>
```

To complete this demonstration, let's add a code to our program that will restart one of the Virtuozzo VEs from the list above.

Restarting a Virtuozzo VE

The `restart` API call from the `vzaenvm` interface is used to restart a Virtuozzo VE. The call accepts a single parameter, the EID of the VE to restart. We will use the EID of one of the VEs from the list that we retrieved in the previous step (see page 39).

```
#XML message. Restarting a VE.
$request=qq~
<packet version="4.0.0" id="4">
  <target>vzaenvm</target>
  <data>
    <vzaenvm>
      <restart>
        <eid>3e25fee2-1163-4336-9e74-8b8097936d47</eid>
      </restart>
    </vzaenvm>
  </data>
</packet>
~;

#Write the XML message to the socket.
print "Restarting a VE...\n\n";
print $socket $request.$/;

#Read the response and display it on the screen.
$buf = <$socket>;
chomp( $buf );
print $buf;
```

If the call succeeds, you should see an output similar to the following:

```
<packet id="4" time="2007-08-29T23:26:50+0000" priority="0"
version="4.0.0">
  <origin>vzaenvm</origin>
  <target>vzclient27-4fce28dd-0cd3-1345-bb94-3192b940fb90</target>
  <dst>
    <director>gend</director>
  </dst>
  <data>
    <vzaenvm>
      <ok/>
    </vzaenvm>
  </data>
  <src>
    <director>gend</director>
  </src>
</packet>
```

The response is a standard VZAgent "OK" message, which is returned when an API call doesn't return any data. It simply means that the call executed successfully.

This concludes the example. Please note that you don't have to log off of VZAgent as this is done automatically when your program exits and the physical connection with VZAgent is terminated.

Summary

In this section, we created a simple client application that did the following:

1. Established a secure SSL connection with VZAgent.
2. Logged on to VZAgent using the system administrator account.
3. Retrieved a list of the Virtuozzo VEs from the Hardware Node.
4. Restarted one of the VEs.

The steps 1 and 2 are the necessary steps that must be taken in any VZAgent application. The steps 2 and 3 have demonstrated how to work with Virtuozzo VEs. The `vzaenvm` interface is not limited to those two tasks of course. You can refer to the **VZAgent XML Programmer's Reference** guide for the complete documentation of the `vzaenvm` interface and its calls.

As you have probably noticed, the program didn't really process the VZAgent response messages (it simply displayed them on the screen). The real application would parse each response to extract the data, and it would also check for errors. In addition all the XML request messages in our program were hard-coded as plain text, which may not be possible and may not be convenient at all times. We will address these and other issues in the later sections of this chapter.

The Complete Program Code

```
#!/usr/bin/perl -w
#
#Copyright (c) 2007 by SWsoft
#

use strict;
use IO::Socket::SSL;

#Message terminating character.
use constant MSG_TERMINATOR => "\0";
$/ = &MSG_TERMINATOR;

#Socket variable
my $socket;

#Create the socket.
#Change the IP address to your Hardware Node address.
print "Connecting to VZAgent...\n\n";
$socket = new IO::Socket::SSL( PeerAddr => '192.168.0.190', PeerPort
=> '4434', Proto => 'tcp' );

#Read the output and display it on the screen.
my $buf = <$socket>;
chomp( $buf );
print $buf;

print "\n";
print "------------------------\n\n";

#XML message. Getting the list of realms.
my $request=qq~
<packet id="2">
  <data>
    <system>
      <get_realm/>
    </system>
  </data>
</packet>
~;

#Write the XML message to the socket.
print "Getting a list of realms...\n\n";
print $socket $request.$/;

#Read the response and display it on the screen.
$buf = <$socket>;
chomp( $buf );
print $buf;
print "\n";
print "------------------------\n\n";

#XML message. Logging in.
#Change the name and password to your administrator name and password.
$request=qq~
<packet version="4.0.0" id="3">
  <data>
    <system>
```

```
      <login>
        <name>cm9vdA==</name>
        <realm>00000000-0000-0000-0000-000000000000</realm>
        <password>MXEydzNl</password>
      </login>
    </system>
  </data>
</packet>
~;

#Write the XML message to the socket.
print "Logging in...\n\n";
print $socket $request.$/;

#Read the response and display it on the screen.
$buf = <$socket>;
chomp( $buf );
print $buf;
print "\n";
print "------------------------\n\n";


#XML message. Getting a list of Virtuozzo VEs.
$request=qq~
<packet version="4.0.0" id="4">
  <target>vzaenvm</target>
  <data>
    <vzaenvm>
      <get_list/>
    </vzaenvm>
  </data>
</packet>
~;

#Write the XML message to the socket.
print "Getting a list of VEs...\n\n";
print $socket $request.$/;

#Read the response and display it on the screen.
$buf = <$socket>;
chomp( $buf );
print $buf;
print "\n";
print "------------------------\n\n";


#XML message. Restarting a VE.
#Change the EID to the EID of your VE.
$request=qq~
<packet version="4.0.0" id="4">
  <target>vzaenvm</target>
  <data>
    <vzaenvm>
      <restart>
        <eid>e9ab2834-ed97-1f4b-bd41-81c27facfc30</eid>
      </restart>
    </vzaenvm>
  </data>
</packet>
~;
```

```
#Write the XML message to the socket.
print "Restarting a VE...\n\n";
print $socket $request.$/;

#Read the response and display it on the screen.
$buf = <$socket>;
chomp( $buf );
print $buf;
print "\n";
print "------------------------\n\n";
```

Advanced Sample Program

In the previous section we created a simple VZAgent client application. Because we wanted to keep things simple and clear, we intentionally used the hard-coded XML requests. For the same purpose, we didn't process VZAgent responses but simply displayed them on the screen. In this section, we will create an application that will demonstrate how to use Perl hash structures to provide more readily available access to the VZagent XML formatted messages.

We will be using the XML-Simple library to encode and decode XML messages in Perl. XML-Simple is a Perl module that allows to store and retrieve structured data in XML. It utilizes the DOM (Document Object Model) concepts allowing to represent an XML message as a binary tree containing strings for XML elements, attributes, and values.

If your Perl installation doesn't have the XML-Simple module installed, you can download it from the following URL:

`http://search.cpan.org/dist/XML-Simple`

The complete documentation for the module is also available at the above location.

Encoding and Decoding an XML message.

The **XML-Simple** module provides two functions: `XMLin()` and `XMLout()`.

`XMLout()` -- takes a data structure (generally a hashref) and returns an XML encoding of that structure. In our sample program, we will initialize a hash reference and populate it with the XML data (a VZAgent request message). We will then use the `XMLout()` function to convert it to an XML string.

`XMLin()` -- parses XML formatted data and returns a reference to a data structure which contains the same information in a more readily accessible form. We will use this function to convert the VZAgent response messages (the XML formatted data) to a hash references.

Consider the following example. Let's say we want to build the `system/login` VZAgent request, which (as you know from the previous section) logs the user in. The XML packet will look similar to the following:

```
<packet version="4.0.0" id="3">
  <data>
    <system>
      <login>
        <name>cm9vdA==</name>
        <realm>00000000-0000-0000-0000-000000000000</realm>
        <password>MXEydzNl</password>
      </login>
    </system>
  </data>
</packet>
```

As you can see in the example above, the values of the name, realm, and password fields are hard-coded into the XML string. What we are going to do instead is initialize a hash reference and populate it with the names of the XML elements, attributes, and values like this:

```
my $hashref = {};
$hashref->{"packet"}[0]{"version"} = "4.0.0";
$hashref->{"packet"}[0]{"id"} = "3";
$hashref->{"packet"}[0]{"data"}[0]{"system"}[0]{"login"}[0]{"name"}[0]
= $name;
$hashref-
>{"packet"}[0]{"data"}[0]{"system"}[0]{"login"}[0]{"realm"}[0] =
$realm;
$hashref-
>{"packet"}[0]{"data"}[0]{"system"}[0]{"login"}[0]{"password"}[0] =
$password;
```

The name, realm, and password values can now be passed to the program as parameters. We can now use the `XMLout()` function to get an XML encoding of that structure, which we can send to VZAgent for processing.

```
my $XMLrequest = XMLout($hashref, keeproot => 1, keyattr => "");
```

The VZAgent response to the login request will look similar to the following example:

```
<packet id="3" priority="0" version="4.0.0">
  <origin>system</origin>
  <target>vzclient19-4fce28dd-0cd3-1345-bb94-3192b940fb90</target>
  <data>
    <system>
      <token>
```

```
        <user>AQUAAAAAIAHdKM5P0wxFE7uUMZK5QPuQAAAAAA==</user>
        <groups>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQAAAAAA==</sid>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQAQAAAA==</sid>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQCgAAAA==</sid>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQAgAAAA==</sid>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQAwAAAA==</sid>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQBAAAAA==</sid>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQBgAAAA==</sid>
          <sid>AQUAAAAAIAHdKM5P0wxFE7uUMZK5QPuQAAAAAA==</sid>
        </groups>
        <deny_only_sids/>
        <privileges/>
      </token>
    </system>
  </data>
</packet>
```


To extract the values from the XML message above, we will first use the `XMLin()` function to convert the XML data to hash reference like this:

```
my $hashref = XMLin($response, keeproot => 1, forcearray => 1, keyattr
=> "");
```

The `$response` variable is a string containing the XML message. The rest of the parameters are the `XMLin()` options (you can find the descriptions of all the options in the the **XML-Simple** documentation). Your program can now extract the values like this:

```
#Extracting the user SID.
my $SID = $hashref-
>{"packet"}[0]{"data"}[0]{"system"}[0]{"token"}[0]{"user"}[0];
print "User SID: $SID\n";

#Extracting the group SIDs.
my $gSID;
my $groupSIDs = $hashref-
>{"packet"}[0]{"data"}[0]{"system"}[0]{"token"}[0]{groups}[0]{sid};
foreach $gSID (@$groupSIDs) {
    print "Group SID: $gSID\n";
}
```

The Complete Program Code

The following sample program takes full advantage of the XML-Simple library and the Perl hash structures to build and parse VZAgent message. A VZAgent request here is prepared and executed using the following steps:

1. A hash reference is initialized which will hold the request message.
2. The `buildXMLHeader` subroutine populates the hash reference tree with the message header data.
3. The message body is then added to the hash reference tree.
4. The hash reference data is converted to an XML document using the `XMLout()` function.
5. The string containing the XML is then submitted to VZAgent.
6. The `readResponseMsg` subroutine reads the VZAgent response message from the socket and converts it to a hash reference.

7 The hash reference data is then checked if it contains a VZAgent error information. If it does, the program is terminated.

8 The response data is extracted from the hash reference and is displayed on the screen.

The program connects to the VZAgent running on the specified server (the IP address is a command line parameter) and logs the specified user in (the name and password are the command line parameters as well). If the login is successful, the program retrieves the list Virtuozzo VEs from the host server. The information retrieved includes the Environment ID (EID), the Virtuozzo-level ID (VEID), the VE hostname, and the VE IP address.

```
#!/usr/bin/perl -w
#
#Copyright (c) 2007 by SWsoft
#

use strict;
use XML::Simple;
use Data::Dumper;
use IO::Socket::SSL;
use MIME::Base64;

#Null-terminating character.
use constant MSG_TERMINATOR => "\0";
$/ = &MSG_TERMINATOR;

#XML version.
use constant XML_VERSION => "1.0";

#Agent protocol version.
use constant VZA_PROTOCOL_VERSION => "4.0.0";

#The System realm ID (authentication database).
use constant SYSTEM_REALM_ID => "00000000-0000-0000-0000-
000000000000";

#Response type constants.
use constant RESPONSE_OK => 0;
use constant RESPONSE_ERROR => 1;
use constant RESPONSE_DATA => 2;

#VZAgent Message ID
my $msgID = 0;

#Socket variable
my $socket;

####################################################

# Subroutine vzaConnect
# Establishes a connection with VZAgent.
# Parameters
#          $address: The server IP address.
#
sub vzaConnect
{
  my $address = shift;
  $socket = new IO::Socket::SSL( PeerAddr => $address, PeerPort =>
'4434', Proto => 'tcp' );
  return RESPONSE_ERROR unless $socket;
}

# Subroutine buildXMLHeader
# Builds the VZAgent XML message header.
# Parameters
```

```
#         $hashref: Hashref to populate.
#         $target:  The name of the target operator.
#
sub buildXMLHeader
{
  my $hashref = shift;
  my $target  = shift;

  $hashref->{"packet"}[0]{"version"} = &VZA_PROTOCOL_VERSION;
  $hashref->{"packet"}[0]{"id"} = $msgID;
  $msgID = $msgID + 1;

  #Add the <target> element to every message except
  #the "system" messages.
  $hashref->{"packet"}[0]{"target"}[0] = $target
      unless $target =~ m/system/;
}

# Subroutine invalidPacket.
# Report the parsing problem and terminates the program.
# Parameters:
#            $element: the missing XML element.
sub invalidPacket
{
  my $element = shift;
  print STDERR "An invalid VZAgent message was received: the $element
element is not present\n";
  exit(1);
}

# Subroutine: checkForError
# Checks if a response message contains
# an error and displays it on the screen if it does.
# Parameters:
#           $hashref: Hashref containing the response.
#           $action:  The operation description.
#
sub checkForError
{
  my $hashref = shift;
  my $action  = shift;

  #If the reponse doesn't have the "packet" and the "data"
  #elements then this is not a valid VZAgent response.
  invalidPacket("packet") if (!$hashref->{"packet"});
  invalidpacket("data") if (!$hashref->{"packet"}[0]{"data"});

  #Read the name of the operator that generated the response.
  my $origin = $hashref->{"packet"}[0]{"origin"}[0];

  #Try reading the error code.
  my $errCode = $hashref-
>{"packet"}[0]{"data"}[0]{$origin}[0]{"error"}[0]{"code"}[0];

  #Try reading the error message.
  my $errMsg  = $hashref-
>{"packet"}[0]{"data"}[0]{$origin}[0]{"error"}[0]{"message"}[0];

  #If there is an error..
  if($errCode){
```

```
      print " \n";
      print "An error occurred while $action: \n";
      print "[$errCode] $errMsg\n\n";
      return RESPONSE_ERROR;
  }
  return 0;
}

# Subroutine readResponseMsg
# Reads an XML response message from the socket and
# converts it to a hashref.
# Returns: Hashref containing the response message.
#
sub readResponseMsg
{
  my $response = <$socket>;
  chomp($response);

  my $hashref = XMLin($response, keeproot => 1, forcearray => 1,
keyattr => "");
  return $hashref;
}

# Subroutine: userLogin
# Logs the user in.
# Parameters:
#            $user:     User name.
#            $password: User password.
#            $realmID:  Realm ID.
#
sub userLogin
{
  my $user     = shift;
  my $password = shift;
  my $realmID  = shift;

  #Initialize the hash structure.
  my $hashref = {};

  #Build the request message header.
  buildXMLHeader($hashref, "system" );

  #Build the message body.
  $hashref-
>{"packet"}[0]{"data"}[0]{"system"}[0]{"login"}[0]{"name"}[0] = $user;
  $hashref-
>{"packet"}[0]{"data"}[0]{"system"}[0]{"login"}[0]{"realm"}[0] =
$realmID;
  $hashref-
>{"packet"}[0]{"data"}[0]{"system"}[0]{"login"}[0]{"password"}[0] =
$password;

  #Convert the hash structure containing the VZAgent
  #request message to an XML document.
  my $XMLrequest = XMLout($hashref, keeproot => 1, keyattr => "");

  #Submit the message to VZAgent.
  print $socket $XMLrequest.$/;

  #Read the response message from VZAgent.
```

```
  $hashref = readResponseMsg();

  #Check if the response contains an error.
  my $rc = checkForError( $hashref, "authenticating the user" );

  #The response contains an error.
  if( $rc == RESPONSE_ERROR ){
    return RESPONSE_ERROR
  }

  #The response contains the data.
  return RESPONSE_DATA;
}

# Subroutine: getVElist
# Retrieves the list of VEs using the
# "get_list" and "get_info" calls from the
# vzaenvm interface.
#
sub getVElist
{
  print "Getting the list of VEs... \n\n";
  my $hashref = {};

  #Build the message header.
  buildXMLHeader($hashref, "vzaenvm");

  #Build the message body.
  $hashref->{"packet"}[0]{"data"}[0]{"vzaenvm"}[0]{"get_list"}[0] =
"";

  #Convert the hash structure to an XML document
  #and submit it to VZAgent.
  #
  my $request = XMLout($hashref, keeproot => 1, keyattr => "");
  print $socket $request.$/;

  #Read VZAgent response.
  $hashref = readResponseMsg();
  my $rc = checkForError($hashref, "getting the list of VEs");

  #Read the entire EID array from the structure into the $eid
variable.
  my $eids = $hashref->{"packet"}[0]{"data"}[0]{"vzaenvm"}[0]{"eid"};
  my $eid;
  my $veid = 0;
  my $hostname = "";
  my $ipaddress = "";
  my $hashref_get_info;

  #Loop through the EID array, read the VE IDs and
  #display them on the screen.
  #
  if($rc == 0){
    foreach $eid (@$eids) {
      print "EID: $eid\n";

      #For each EID, retrieve the additional VE info using the
      #"get_info" call from the "vzaenvm" interface.
      #
```

```
      $hashref_get_info = {};
      buildXMLHeader($hashref_get_info, "vzaenvm");

      #The "get_info" call accepts the Environment ID (EID) as a
parameter.
      $hashref_get_info-
>{"packet"}[0]{"data"}[0]{"vzaenvm"}[0]{"get_info"}[0]{"eid"}[0] =
$eid;
      $request = XMLout($hashref_get_info, keeproot => 1, keyattr =>
"" );
      print $socket $request.$/;

      #Read VZAgent reponse.
      $hashref_get_info = readResponseMsg();
      $rc = checkForError($hashref_get_info, "getting the VE info" );

      #The response contains the detailed VE information.
      #We are reading and displaying on the screen just the
      #Virtuozzo-level VE ID (VEID), the hostname, and the IP address.
      #
      $veid = $hashref_get_info-
>{"packet"}[0]{"data"}[0]{"vzaenvm"}[0]{"env"}[0]{"virtual_config"}[0]
{"veid"}[0];
      $hostname = $hashref_get_info-
>{"packet"}[0]{"data"}[0]{"vzaenvm"}[0]{"env"}[0]{"virtual_config"}[0]
{"hostname"}[0];
      $ipaddress = $hashref_get_info-
>{"packet"}[0]{"data"}[0]{"vzaenvm"}[0]{"env"}[0]{"virtual_config"}[0]
{"net_device"}[0]{"ip_address"}[0]{"ip"}[0];
      print "VEID: $veid\n";
      print "Hostname: $hostname\n";
      print "IP Address: $ipaddress\n\n";
    }
  }
}

##################################################
#                                                #
#    Main program                                #
#                                                #
##################################################

if (!$ARGV[2]) {
  print STDERR "Usage: VZAgentExample2.pl ip_address username
password\n";
  exit(1);
}

#Read the user input.
#Convert the user name and password to base64.
my $address   = $ARGV[0];
my $user      = encode_base64( $ARGV[1], '' );
my $password  = encode_base64( $ARGV[2], '' );

#Connect to Agent.
print "\nConnecting to Agent...\n";
my $rc = vzaConnect( $address );

if( $rc == RESPONSE_ERROR ){
  die "Cannot establish connection with Agent.\n";
```

```
}

#Read VZAgent response.
readResponseMsg();
print "Connection established.\n\n";

#Log the user in.
#We are passing the system realm ID because we are
#assuming that the user will be logging in as the system
#administrator. In a real application, we would either get
#the realm ID as a parameter from the user, or we could
#get the list of realms from VZAgent and would try to
#authenticate the user against every single one of them
#until the login is successful.
#
print "Logging in...\n";
$rc = userLogin($user, $password, &SYSTEM_REALM_ID );

if( $rc == RESPONSE_ERROR ){
  exit 0;
}

print "Login successful.\n\n";

#Retrieve the VE list from the Hardware Node.
getVElist();

exit 0;
```

Some Programming Techniques

When you program with the VZAgent XML protocol using DOM (Document Object Model), there is usually a limited set of functions providing full access to an XML message. No matter which programming language you use, if you have this set of functionality, you are ready to work with VZAgent packets. The following is a typical list of such functions (the function names that we use here are not the exact names, different programming languages use their own function names).

Encoding -- any function that converts an XML text to a binary tree, like the `XMLin()` function that we used in our sample program.

Decoding -- any function converting a binary tree structure to an XML text, like the `XMLout()` function in our example.

Seek and Seek Up -- a way of navigating through XML binary trees. In Perl, this is done by specifying hash elements like `{"packet"}[0]{"data"}[0]{"system"}[0]{"login"}[0]{"name"}[0]`. In other programming languages it can be implemented as a function iterating some pointer in an XML binary tree structure. In this case it might look like `seek("packet/data/system/login/name")`, or simply `seek_up()` to navigate upwards from the current position.

New child -- any function adding a new child node to the current node or the specified node of the tree. In our sample, this is done by simply including these nodes in the hash reference definition like this:

```
{"packet"}[0]{"data"}[0]{"system"}[0]{"login"}[0]{"realm"}[0] =
$realm;
```

Your programming language may have a function similar to the following:

```
newChild("packet", "data")
```

First child -- returns a pointer to the first child of the current or specified node in order to start iterating through its children. In our sample this function takes a form of assigning the entire array of the node's child elements to another hash:

```
my $veids = $XMLOUT->{"packet"}[0]{"data"}[0]{"hwm"}[0]{"veid"};
```

and then we iterate through it using `foreach my $veid (@$veids) { }`

Next sibling -- iterates to the next sibling of the current node.

Put value -- any function assigning a value to the specified node. The simple assignment used in our sample

```
$XMLContainer-
>{"packet"}[0]{"data"}[0]{"vem"}[0]{'stop'}[0]{'veid'}[0]=$veid;
```

can be replaced with a function call in other languages. Something like:

```
putValue("packet/data/vem/stop/veid", veid)
```

or

```
putValue(curNode, veid)
```

Get value -- any function returning the value of the specified node. The idea here is the same as for putting a value:

```
veid = getValue("packet/data/vem/stop/veid");
```

or

```
veid = getValue(curNode);
```

Put attribute -- assigning a value to the specified attribute is pretty much similar to assigning a value to an element:

```
$XMLContainer->{"packet"}[0]{"version"} = &AGENT_PROTOCOL_VERSION;
```

or

```
putAttribute("packet/version", AGENT_PROTOCOL_VERSION);
```

Get attribute -- retrieving the value of an attribute. This is much like retrieving the value of an element.

Now, when creating a message, you would use the New child, Put value, and Put attribute functions and Decode before sending it to VZAgent, whereas for message parsing, the Seek, First child, Next sibling, Get value, and Get attribute functions, would be of use after Encoding is done.

Please note that the functions discussed above are not the exact names of some programming language functions, they simply constitute the minimal functionality set allowing you to work with the VZAgent XML protocol.

XML Libraries

There is a number of existing third-party software libraries for different languages which can be used for XML encoding and decoding. Some of the well-known and extensively used XML parsers are the following:

Expat -- an XML parser library written in C. It is a stream-oriented parser in which an application registers handlers for things the parser might find in the XML document (like start tags). (http://expat.sourceforge.net)

Libxml -- this is the XML C library developed for the Gnome project. Although it is written in C, a variety of language bindings make it available in other environments. (http://xmlsoft.org)

Xerces -- XML parsers in Java and C++ (with Perl and COM bindings). (http://xml.apache.org)

MSXML -- Microsoft XML Parser. (http://msdn.microsoft.com/xml)

XML-Simple -- an easy API to read and write XML in Perl. (http://search.cpan.org/dist/XML-Simple)

For the general information on XML and XML Schema please refer to the http://www.w3c.org site.

Login and Session Management

The VZAgent login procedure comprises the following steps:

1. Getting a list of realms from VZAgent and choosing the realm against which to authenticate the user. You can perform this operation without being authenticated by VZAgent. See the **Authentication Concepts** section (see page 14) for the explanation of what a realm is.
2. Log in using the name and password of the user that exists in the selected realm. If authentication is successful, a permanent connection session will be created for the user.
3. Create an additional, temporary session for the user if desired and use that session to execute VZAgent requests.

The rest of this section describes each step in detail.

Retrieving the Realm Information

To retrieve the list of the existing realms, use the following VZAgent request:

```
<packet version="4.0.0" id="2">
  <data>
    <system>
      <get_realm/>
    </system>
  </data>
</packet>
```

We already demonstrated how it's done earlier in this guide (see page 36). This call does not require you to be logged in (please don't confuse it with another `get_realm` call that belongs to the `authm`, authentication management, interface). The VZAgent response will contain the list of the available realms:

```
<packet xmlns:ns2="http://www.swsoft.com/webservices/vzl/4.0.0/types"
xmlns:ns1="http://www.swsoft.com/webservices/vzl/4.0.0/dirm"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
id="8c46e79de1t18ber68c" priority="0" version="4.0.0">
<origin>system</origin>
<target>vzclient121-4fce28dd-0cd3-1345-bb94-3192b940fb90</target>
<data>
  <system>
    <realms>
      <realm xsi:type="ns1:dir_realmType">
        <login>
          <name>Y249dnphZ2VudCxkYz1WWkw=</name>
          <realm>458d583f-f2d8-7940-a9d3-a9a3d2ec1509</realm>
        </login>
        <builtin/>
        <name>Virtuozzo Internal</name>
        <type>1</type>
        <id>458d583f-f2d8-7940-a9d3-a9a3d2ec1509</id>
        <address>vzsveaddress</address>
        <port>389</port>
        <base_dn>ou=4fce28dd-0cd3-1345-bb94-
3192b940fb90,dc=vzl</base_dn>
        <default_dn>cn=users,ou=4fce28dd-0cd3-1345-bb94-
3192b940fb90,dc=vzl</default_dn>
      </realm>
      <realm xsi:type="ns2:realmType">
        <builtin/>
        <name>System</name>
        <type>0</type>
        <id>00000000-0000-0000-0000-000000000000</id>
      </realm>
      <realm xsi:type="ns2:realmType">
        <builtin/>
        <name>Virtuozzo VE</name>
        <type>1000</type>
        <id>00000000-0000-0000-0100-000000000000</id>
      </realm>
    </realms>
  </system>
</data>
</packet>
```

The result set contains three realm entries: Virtuozzo Internal, System, and Virtuozzo VE. The following describes each one in detail.

Virtuozzo Internal realm

```
<realm xsi:type="ns1:dir_realmType">
  <login>
    <name>Y249dnphZ2VudCxkYz1WWkw=</name>
    <realm>458d583f-f2d8-7940-a9d3-a9a3d2ec1509</realm>
  </login>
  <builtin/>
  <name>Virtuozzo Internal</name>
  <type>1</type>
  <id>458d583f-f2d8-7940-a9d3-a9a3d2ec1509</id>
  <address>vzsveaddress</address>
  <port>389</port>
  <base_dn>ou=4fce28dd-0cd3-1345-bb94-3192b940fb90,dc=vzl</base_dn>
  <default_dn>cn=users,ou=4fce28dd-0cd3-1345-bb94-
3192b940fb90,dc=vzl</default_dn>
</realm>
```

The Virtuozzo Internal realm is an authentication database that is installed on the host server during Virtuozzo installation. This database is used to store the Virtuozzo and VZAgent specific authentication information. Let's take a look at the XML structure above. The type of the `realm` element is `dir_realmType`. It is a descendant of the base `realmType` type and it is used to hold the information about an LDAP-compliant directory. The `type` element specifies the realm type -- the value of `1` (one) means LDAP directory. The `name` element inside the `login` node is the user name that VZAgent will use to bound to the directory instance. The name, in this case, is a distinguished name (DN) identifying the user object in the directory. The user password is hidden from you. VZAgent uses this information to bound to the directory to perform user authentication. The empty `builtin` element indicates that this is a built-in Virtuozzo realm (as opposed to custom realms created by users). In fact, the rest of the realms in this example are built-in realms -- the realms that come preinstalled with VZAgent. The `address`, `port`, `base_dn`, and `default_dn` parameters describe the directory in terms of connectivity. Again, all of these elements are used by VZAgent to bound to the directory instance and, at this point, are of little interest to us. The `id` element contains the realm ID. This is the ID that you will use in all other calls that require it, such as the `login` call that will be described later in this section. Please note that the ID of the Internal Virtuozzo realm in your Virtuozzo installation may not be the same as the ID in our example. There can be only one Virtuozzo Internal realm. To obtain its ID, simply search for the realm of type `1` (LDAP directory) and for the presence of the `builtin` element.

System realm

```
<realm xsi:type="ns2:realmType">
  <builtin/>
  <name>System</name>
  <type>0</type>
  <id>00000000-0000-0000-0000-000000000000</id>
</realm>
```

The System realm represents the host operating system user registry. When VZAgent is first installed, you will not have any VZAgent-specific users in any of the other realms except the System realm. If you have just started with VZAgent, use the system administrator account to log in to it. VZAgent knows how to identify the user with system administrator privileges and by default grants her/him unlimited access to the host server and all Virtuozzo VEs hosted by it. The ID of the System realm in your installation will probably be the same as in this example (all zeros) but it is not guaranteed, so you should retrieve and obtain it from the VZAgent installed on your server. You find the System realm record in the result set by its type, which should be `0` (zero).

Virtuozzo VE realm

```
<realm xsi:type="ns2:realmType">
  <builtin/>
  <name>Virtuozzo VE</name>
  <type>1000</type>
  <id>00000000-0000-0000-0100-000000000000</id>
</realm>
```

The Virtuozzo VE realm represents an operating system user registry inside a VE. Use this realm if you would like to log in to VZAgent as an operating system user of one of the VEs that is installed on the Hardware Node. Once again, the ID of this realm in your Virtuozzo installation may not be the same as the ID you see in the example above. Always get the realm ID from the VZAgent installed on your server.

Logging In

We've demonstrated the login procedure earlier when we created a sample program (see page 36). This subsection describes it in detail.

The initial login is performed by executing the `system.login` request as shown below:

```
<packet version="4.0.0" id="3">
  <data>
    <system>
      <login>
        <name>cm9vdA==</name>
        <realm>00000000-0000-0000-0000-000000000000</realm>
        <password>MXEydzNl</password>
      </login>
    </system>
  </data>
</packet>
```

In this example, we are logging in as the `root` user (the name and the password values are base-64 encoded according to the XML Schema). We are specifying the ID of the system realm that we retrieved earlier because `root` is the user of the host server (the Hardware Node). As a result, VZAgent will find the user in the host operating system registry and will verify that the supplied credentials are correct. If were wanted to log in as a user from the Virtuozzo Internal realm, we would execute the same call supplying the appropriate user name and password but passing the ID of the Virtuozzo Internal realm (which we also retrieved earlier).

When logging in as a user from the Virtuozzo VE realm, the call is executed slightly differently. Let's say that we want to log in to VZAgent as the `root` user from one of the VEs running on the Hardware Node. This is how you do it:

```
<packet version="" id="3">
  <data>
    <system>
      <login>
        <name>cm9vdA==</name>

<domain>ZTlhYjI4MzQtZWQ5Ny0xZjRiLWJkNDEtODFjMjdmYWNmYzMw</domain>
        <realm>00000000-0000-0000-0100-000000000000</realm>
        <password>MXEydzNl</password>
      </login>
    </system>
  </data>
</packet>
```

Note that compared to the previous login example, the XML packet above contains the additional `domain` element. When logging in as a user from the Virtuozzo VE realm, the `domain` element must contain the VZAgent-level ID (EID) of the Virtual Environment. See **Retrieving a List of Virtuozzo VEs** section (on page 39) for an example on how to retrieve the EIDs.

If the login was successful, the output will contain the user security information and will look similar to the following:

```
<packet id="3" priority="0" version="4.0.0">
  <origin>system</origin>
  <target>vzclient19-4fce28dd-0cd3-1345-bb94-3192b940fb90</target>
  <data>
```

```
    <system>
      <token>
        <user>AQUAAAAAIAHdKM5P0wxFE7uUMZK5QPuQAAAAAA==</user>
        <groups>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQAAAAAA==</sid>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQAQAAAA==</sid>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQCgAAAA==</sid>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQAgAAAA==</sid>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQAwAAAA==</sid>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQBAAAAA==</sid>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQBgAAAA==</sid>
          <sid>AQUAAAAAIAHdKM5P0wxFE7uUMZK5QPuQAAAAAA==</sid>
        </groups>
        <deny_only_sids/>
        <privileges/>
      </token>
    </system>
  </data>
</packet>
```

The output contains the security IDs (SIDs) of the user and all the groups that the user belongs to.

Sessions

When you execute the `system/login` call, a permanent session is created for the user whose credentials were included in the request. A permanent session never expires and it becomes associated with the physical connection to VZAgent that your program is using. If you are not planning on logging in multiple users from the same program, you may simply use this session to execute all your request. When you are done working with VZAgent, you may simply exit and the session will be terminated automatically.

Optionally, you may create additional sessions using the `sessionm` API interface. This interface allows to log in additional clients or create additional sessions for the clients that are already logged in. Please note that you can use the `sessionm` interface only after you've logged in using the `system/login` call. The following request logs the user in and creates a temporary session:

```
<packet version="4.0.0" id="2">
  <target>sessionm</target>
  <data>
    <sessionm>
      <login>
        <name>cm9vdA==</name>
        <realm>00000000-0000-0000-0000-000000000000</realm>
        <password>bXlwYXNz</password>
        <expiration>1200</expiration>
      </login>
    </sessionm>
  </data>
</packet>
```

The parameters in this call are used similarly to the `session/login` call described in the previous section. The output will contain the SIDs plus the ID of the session that was created:

```
<packet id="2" time="2007-09-10T09:42:13+0000" priority="0"
version="4.0.0">
  <origin>sessionm</origin>
  <target>vzclient133-4fce28dd-0cd3-1345-bb94-3192b940fb90</target>
  <dst>
    <director>gend</director>
  </dst>
  <data>
    <sessionm>
      <session_id>vzl.40000.4.4fce28dd-0cd3-1345-bb94-
3192b940fb90..f7c46e51175t321d6069r202d</session_id>
      <token>
        <user>AQUAAAAAIAHdKM5P0wxFE7uUMZK5QPuQAAAAAA==</user>
        <groups>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQAAAAAA==</sid>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQAQAAAA==</sid>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQCgAAAA==</sid>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQAgAAAA==</sid>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQAwAAAA==</sid>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQBAAAAA==</sid>
          <sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQBgAAAA==</sid>
          <sid>AQUAAAAAIAHdKM5P0wxFE7uUMZK5QPuQAAAAAA==</sid>
        </groups>
        <deny_only_sids/>
        <privileges/>
      </token>
```

```
    </sessionm>
  </data>
  <src>
    <director>gend</director>
  </src>
</packet>
```

The `session_id` element inside the `sessionm` node contains the new session ID. When sending requests, you must include the session ID in every VZAgent request. Failure to do so will result in the message being sent using the default session created by the `system/login` call. The following example shows how to include the session ID in a VZAgent request message.

```
<packet version="4.0.0" id="23">
  <session>your_session_id_goes_here</session>
  <data>
    ............
  </data>
</packet>
```

User sessions expire after some predefined period of inactivity or after the timeout limit specified in the `expiration` parameter is reached. The default session timeout value is specified in the VZAgent configuration. If the expiration element is included in the request then its value overrides the default timeout value. Each request sent while a temporary session is still active resets the session timeout to its initial state.

Creating and Configuring a Virtuozzo VE

Getting a List of Sample Configurations

When creating a Virtuozzo VE, you must choose a sample configuration for it. Virtuozzo comes with a number of sample configurations that are installed on the host server at the time the Virtuozzo software is installed. To retrieve the list of the available configurations, use the `env_samplem/get_sample_conf` request as shown in the following example:

```
<packet version="4.0.0" id="23">
  <target>env_samplem</target>
  <data>
    <env_samplem>
      <get_sample_conf/>
    </env_samplem>
  </data>
</packet>
```

The output will contain all available configurations with the complete set of parameters for each one. You can review the parameters and their values but when creating a VE, what you really want to look at is the configuration name and ID. The following example shows the typical output. The QoS (quality of service) and some of the other parameters are omitted for brevity in our example. The `id` (sample configuration ID), `name` (configuration name), and `version` (the platform, architecture, and virtualization technology information) are highlighted in bold in the example for your convenience:

```
<packet xmlns:ns2="http://www.swsoft.com/webservices/vzl/4.0.0/types"
xmlns:ns1="http://www.swsoft.com/webservices/vzl/4.0.0/env_samplem"
xmlns:ns3="http://www.swsoft.com/webservices/vza/4.0.0/vzatypes"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
id="17c46e7e9f0t6df1r68c" time="2007-09-10T10:03:43+0000" priority="0"
version="4.0.0">
<origin>env_samplem</origin>
<target>vzclient122-4fce28dd-0cd3-1345-bb94-3192b940fb90</target>
<dst>
  <director>gend</director>
</dst>
<data>
  <env_samplem>
    <sample_conf xsi:type="ns2:sample_confType">
      <env_config xsi:type="ns3:venv_configType">
        <offline_management>1</offline_management>
        <architecture>i386</architecture>
        <os xsi:type="ns2:osType">
          <platform>linux</platform>
          <name/>
        </os>
        <slm_mode>all</slm_mode>
        <type>virtuozzo</type>
      </env_config>
      <id>c607f3c6-16b3-214a-9079-8113fdfa1630</id>
      <name>slm.256MB</name>
      <vt_version>
```

```
      <platform>Linux</platform>
      <architecture>i386</architecture>
      <vt_technology>virtuozzo</vt_technology>
    </vt_version>
  </sample_conf>
  <sample_conf xsi:type="ns2:sample_confType">
    <env_config xsi:type="ns3:venv_configType">
      <on_boot>0</on_boot>
      <offline_management>1</offline_management>
      <architecture>i386</architecture>
      <os xsi:type="ns2:osType">
        <platform>linux</platform>
        <name/>
      </os>
      <type>virtuozzo</type>
    </env_config>
    <id>01cb5525-d247-3f45-aa47-0d19eb8285b5</id>
    <name>confixx</name>
    <vt_version>
      <platform>Linux</platform>
      <architecture>i386</architecture>
      <vt_technology>virtuozzo</vt_technology>
    </vt_version>
  </sample_conf>
  <sample_conf xsi:type="ns2:sample_confType">
    <env_config xsi:type="ns3:venv_configType">
      <offline_management>1</offline_management>
      <architecture>i386</architecture>
      <os xsi:type="ns2:osType">
        <platform>linux</platform>
        <name/>
      </os>
      <slm_mode>all</slm_mode>
      <type>virtuozzo</type>
    </env_config>
    <id>c2692640-065c-644c-94cc-1dceb42e16c5</id>
    <name>slm.2048MB</name>
    <vt_version>
      <platform>Linux</platform>
      <architecture>i386</architecture>
      <vt_technology>virtuozzo</vt_technology>
    </vt_version>
  </sample_conf>
  <sample_conf xsi:type="ns2:sample_confType">
    <env_config xsi:type="ns3:venv_configType">
      <on_boot>0</on_boot>
      <offline_management>1</offline_management>
      <architecture>i386</architecture>
      <os xsi:type="ns2:osType">
        <platform>linux</platform>
        <name/>
      </os>
      <type>virtuozzo</type>
    </env_config>
    <id>b048bcc2-c80c-6d42-9e6d-ffe808d6a83c</id>
    <name>basic</name>
    <vt_version>
      <platform>Linux</platform>
      <architecture>i386</architecture>
      <vt_technology>virtuozzo</vt_technology>
```

```
      </vt_version>
    </sample_conf>
  </env_samplem>
</data>
</packet>
```

After executing this request, select the sample configuration that you would like to use and extract its ID from the response message for future reference (you will use it in the request that will create the VE).

Getting a List of OS Templates

A Virtuozzo VE is based on an Operating System template (OS template). The OS templates are shipped with Virtuozzo and are installed on the Hardware Node during the Virtuozzo installation. To get the list of the available OS templates, use the `vzapkgm/list` call as shown in the following example:

```
<packet version="4.0.0" id="32">
  <target>vzapkgm</target>
  <data>
    <vzapkgm>
      <list>
        <options>
          <type>os</type>
        </options>
      </list>
    </vzapkgm>
  </data>
</packet>
```

The output will contain the list of the available OS templates:

```
<packet id="32" time="2007-09-10T10:22:45+0000" priority="0"
version="4.0.0">
  <origin>vzapkgm</origin>
    <target>vzclient136-4fce28dd-0cd3-1345-bb94-3192b940fb90</target>
    <dst>
      <director>gend</director>
    </dst>
    <data>
      <vzapkgm>
        <packages>
          <package>
            <name>redhat-as3-minimal</name>
            <version>20061020</version>
            Linux
              <platform>Linux</platform>
              <name/>
            </os>
            <arch>x86</arch>
            <os_template>1</os_template>
            <cached>1</cached>
            <uptodate>0</uptodate>
            <technology>nptl</technology>
            <technology>x86</technology>
            <base>1</base>
          </package>
        </packages>
      </vzapkgm>
    </data>
</packet>
```

Select the OS template that you would like to use and get its name from the response. The template name will be used as one of the parameters in the call that will create the VE later. In our example, we have just one template and its name is "redhat-as3-minimal".

Populating the VE Configuration Structure

After you've selected the configuration sample and the OS template, you have to populate the VE configuration structure with these and other values. The most commonly used and important parameters are described in the following table:

Parameter	Description
base_sample_id	The sample configuration ID.
os_template/name	The OS template name.
name	The VE computer name.
hostname	The VE hostname.
veid	Virtuozzo-level VE ID. This can be any integer number greater than 100.
on_boot	Start VE automatically on system boot.
offline_management	Enable the "offline-management" feature for the VE.
ip_address	The VE IP address. In the example that will follow, we will assign the IP address to the default venet0 virtual network adapter. The venet0 adapter is created automatically for every VE. We could also create our own virtual network adapter inside a VE and customize it according to our needs. For more info on how to create and configure virtual ethernet adapters, see the venv_configType and net_vethType type specifications in the **VZAgent XML API Reference** guide.

The rest of the configuration parameters (such as disks quota, CPU parameters, etc.) can also be customized but it should only be done by the experienced Virtuozzo users. In this example, we will set all of the parameters from the table above, so the configuration portion of the XML request that will create our VE will look like this:

```
<config>
  <name>My-VE10</name>
  <hostname>Host-110</hostname>
  <base_sample_id>c607f3c6-16b3-214a-9079-
8113fdfa1630</base_sample_id>
  <veid>110</veid>
  <on_boot>true</on_boot>
  <offline_management>true</offline_management>
  <os_template>
    <name>redhat-as3-minimal</name>
  </os_template>
  <net_device>
    <id>venet0</id>
    <ip_address>
      <ip>10.17.3.121</ip>
    </ip_address>
    <host_routed/>
  </net_device>
</config>
```

You can use your own name, hostname, veid, and IP address of course.

Creating the Virtual Environment

The final step in creating a Virtuozzo VE is to build the XML request and send it to VZAgent. The following is an example of such request:

```
<packet version="4.0.0" id="2">
  <target>vzaenvm</target>
  <data>
    <vzaenvm>
      <create>
        <config>
          <name>My-VE10</name>
          <hostname>Host-110</hostname>
          <base_sample_id>c607f3c6-16b3-214a-9079-
8113fdfa1630</base_sample_id>
          <veid>110</veid>
          <on_boot>true</on_boot>
          <offline_management>true</offline_management>
          <os_template>
            <name>redhat-as3-minimal</name>
          </os_template>
          <net_device>
            <id>venet0</id>
            <ip_address>
              <ip>10.17.3.121</ip>
            </ip_address>
            <host_routed/>
          </net_device>
        </config>
      </create>
    </vzaenvm>
  </data>
</packet>
```

If the VE is created successfully, you should see the output similar to the following:

```
<packet id="2" time="2007-09-10T11:02:33+0000" priority="4000"
version="4.0.0">
  <origin>vzaenvm</origin>
  <target>vzclient139-4fce28dd-0cd3-1345-bb94-3192b940fb90</target>
  <dst>
    <director>gend</director>
  </dst>
  <data>
    <vzaenvm>
      <env>
        <parent_eid>00000000-0000-0000-0000-000000000000</parent_eid>
        <eid>8d5c125b-e7f5-c448-9c8a-ee7ccab18599</eid>
        <status>
          <state>1</state>
        </status>
        <alert>0</alert>
        <config/>
        <virtual_config>
          <veid>110</veid>
          <type>virtuozzo</type>
        </virtual_config>
      </env>
    </vzaenvm>
```

```
  </data>
</packet>
```

The output contains the EID that was assigned to the new VE by VZAgent, the EID of the parent Environment (the Hardware Node), and some of the VE information. To verify that the VE was actually created, you can log in to your Hardware Node and run the `vzlist` command from the command prompt. You should see the VE in the list.

Configuring a Virtuozzo VE

Retrieving a VE Configuration

Destroying a Virtuozzo VE

Performance Monitoring

Events and Alerts

CHAPTER 4

Using SOAP API

The material in this chapter is intended for the developers who would like to develop client applications using the SOAP API. To use this documentation productively, you should have a basic idea of what SOAP is, some general programming experience, and a knowledge of one of the programming languages such as C#. We also assume that you are comfortable working with XML and have some experience working with XML Schema language (also referred to as XML Schema Definition or XSD).

In This Chapter

Introduction 71
Creating a Simple Client 73
Invoking Web Services 77
Other SOAP Clients and Their Known Issues 86
Using SOAP API 87
Troubleshoting 96

Introduction

This section provides an introduction to the VZAgent SOAP API.

Overview

VZAgent SOAP API is based on an open standards like SOAP and WSDL. With SOAP API, you build your client applications using one of the third-party development tools that can generate client code from WSDL specifications. The code generated from WSDL documents is a set of objects in your application's native programming language. You work with data structures using object properties and you make API calls by invoking object methods.

The SOAP API shares the XML schema with the XML API. Please refer to **Using XML API** chapter (on page 18) for general information on the XML schema, a detailed description of the XML API request and response packets, and other important information.

Key Features

The following describes the key SOAP API features:

- Supports the full set of the VZAgent on-demand functionality, which covers the whole range of the services provided by VZAgent.
- Supports both the SOAP 1.1 and SOAP 1.2 protocols.
- Provides WSDL schemas for automatic code generation.
- Supports a variety of third-party SOAP clients, including Microsoft .NET Framework 1.1.

Limitations

The SOAP API in VZAgent protocol version 4.0.0 has the following limitations:

- Operates only over the HTTPS protocol.
- Supports only the on-demand functionality (no notifications are available).

The Location of WSDL

To use the SOAP API, you will have to generate the client code (the interfaces and proxies) from WSDL specifications. The WSDL is located at the following URL:

`http://www.swsoft.com/webservices/vza/`*Version*`/VZA.wsdl`

Where *Version* is the VZAgent protocol version number. The URL to version 4.0.0 is as follows:

`http://www.swsoft.com/webservices/vza/4.0.0/VZA.wsdl`

Generating Client Code from WSDL

The SOAP client that you are using should have a documentation describing how to generate the interfaces and proxies from WSDL. Please follow the instructions and supply the URL of the VZAgent WSDL.

If you are using Microsoft Visual Studio .NET, then you will find the complete instructions on how to generate and use the code later in this guide.

Installation

Server side

To access the VZAgent Web services, you need VZAgent software installed on your server. Please refer to the **Installation and Upgrade** section (on page 10) for the VZAgent installation instructions.

Client side

All you need on the client side is a third-party SOAP client, such as Microsoft .NET Framework. If you want to develop your programs using one of the scripting languages (Perl, PHP, etc.), then you will need an interpreter or compiler with SOAP support. No other software of any kind is required.

Creating a Simple Client

This section walks though the basics of creating a simple client application. The application will execute a single SOAP API call, which will obtain and return to the client the date and time from the server.

We will use Microsoft Visual Studio .NET and will write our program in C#.

Step 1: Choosing a Development Project

You can choose any type of Visual Studio .NET C# project for your application. Your choice depends on your application requirements only. For our simple example, let's select a C# Windows console application and call it `CSSoap`.

1. In Microsoft Development Environment, select **File** > **New** > **Project**.
2. In the **New Project** window, select **Visual C# Projects** and then select the **Console Application** template.
3. Enter `CSSoap` as a name for your project, choose a location for your project files and click **OK**.

Step 2: Generating the Stubs from WSDL

1. In **Solution Explorer**, select the `CSSoap` project.
2. On the **Project** menu, select **Add Web Reference**. The **Add Web Reference** window opens.
3. In the **URL** field, type (or copy and paste) this URL `http://www.swsoft.com/webservices/vza/4.0.0/VZA.wsdl`
4. Press the **Go** button. You should see a single entry in the **Web services found at this URL** list: `1 Service Found: VZA`.

5 Type the desired name for the web service in the **Web reference name** field. This name will be used in your code as the C# namespace to access the selected service. For our example, let's name it `VZA`.

6 Press the **Add Reference** button. This generates stubs from WSDL and adds them to the project. A new item `VZA` appears in **Solution Explorer** in the **Web References** folder. If you switch to the **Class View** tab in **Solution Explorer**, you will see the generated stubs there.

You can now start using generated classes to access VZAgent services.

Step 3: Adding the Code

At the beginning, you have the generated code in the `Class1.cs` file looking like this:

Class1.cs

```
using System;
namespace CSSoap
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            //
            // TODO: Add code to start application here
            //
            Console.Read();
        }
    }
}
```

We added `Console.Read()` at the end of `Main()` so that our console window stays open until a key is pressed.

Certificates Policy Preparation

Since VZAgent SOAP uses HTTPS as a transport protocol, we have to deal with the certificate issues. For the purpose of this example, we're going to use the "trust all certificates" policy. We'll create a class that implements such a policy for us and passes it to the certificate policy manager during logon. Here is the class:

```
// Class TrustAllCertificatePolicy.
// Used as a certificate policy provider.
// Allows all certificates.
public class TrustAllCertificatePolicy : System.Net.ICertificatePolicy
{
    public TrustAllCertificatePolicy()
    {}
    public bool CheckValidationResult(System.Net.ServicePoint sp,
    System.Security.Cryptography.X509Certificates.X509Certificate
cert,
    System.Net.WebRequest req, int problem)
    {
        // Trust all certificates.
        return true;
    }
}
```

Connection URL

The VZAgent server listens for SOAP requests on port 4646. The connection URL, therefore, will look similar to the following example:

```
https://192.168.0.218:4646
```

Naturally, you will have to use the actual IP address of your server (the server where you have VZAgent software installed and running).

Logging in and Creating a Session

Before we can start sending requests to VZAgent, we must establish a connection with it, let it authenticate us using the credentials that we are going to supply, and create a session for our connection.

```
// Method, provides initial login to url with user and password
private string Login(string url, string user, string password)
{
    ............
}
```

The above function will establish a connection to VZAgent and will attempt to authenticate the user using the supplied credentials. If the credentials are valid, the function will return a session ID. We'll invoke the function from `Main()` and print out the result. The sequence of our operations will be invoked from the `Run()` function:

```
// session ID.
private string m_sid;
// URL
private string m_url;

// Main Function for our test.
public void Run()
{
    ......
}
```

Here, we introduce the fields `m_sid` and `m_url` that we will use to store the connection data for future operations. Let's also add exception handling to `Main()` and print out the exception text to the console:

```
static void Main(string[] args)
{
    //
    // TODO: Add code to start application here
    //
    Class1 VZAServer = new Class1();
    try {
        VZAServer.Run();
    }
    catch(System.Web.Services.Protocols.SoapException ex)
    {
        Console.WriteLine(ex.Code.ToString() + ", " + ex.Message);
        Console.WriteLine("Details:" + ex.Detail.InnerText);
    }
    catch(System.Xml.XmlException xmlex)
    {
        Console.WriteLine(xmlex.ToString());
    }
    catch(System.InvalidOperationException opex)
    {
        Console.WriteLine(opex.Message + "\n" + opex.InnerException);
    }
    Console.Read();
}
```

Step 4: Running the Sample

Now you can build and run the program.

From the main menu, select **Build** and then **Build Solution**. Then select **Debug** > **Start** (or **Start without Debugging**) to run the sample. You should see the following output on the screen:

```
Session ID: vzagent0.30000.2c41b96ceet3768e77b
```

Invoking Web Services

This section provides useful information that will help you make your development efforts as trouble-free as possible. Some of the material presented here will also help you to overcome certain problems that may arise due to differences in SOAP client implementations for different platforms.

SOAP Bindings

The .NET SOAP client generates the key classes with the `Binding` postfix, e.g. `filemBinding`, `envmBinding`, `semBinding`, etc. Each of them contains methods provided by the equivalent VZAgent XML interface: `filem`, `envm`, `sem` respectively. The **Virtuozzo SDK XML Reference** guide can be used as a knowledge base for the existing VZAgent services. Please keep in mind that only the on-demand calls are supported in the SOAP API at the moment.

Optional Elements

Many parameters that you supply to VZAgent API calls or receive from VZAgent are defined in the XML schema as optional elements. In XML calls, this means that you include an element or omit it depending on the task that you are trying to perform. In response messages, an optional element may be similarly included or not.

Unlike XML optional elements, the class members in traditional programming languages cannot be "optional" and therefore are handled differently in this respect. In generated SOAP code, you can encounter optional elements as primitive types (`int`, `bool`, etc.), complex types (strings, classes, structures), or arrays. The following describes how to handle each element type in your code.

Primitive Types

A primitive type element is usually flagged by a corresponding `bool` element declared just below it. The name of the latter is made of the name of the former with an added `Specified` suffix.

As an illustration, let's take a look at the `userType` class.

```
public class userType {

    /// <remarks/>
    public userTypeInitial_group initial_group;

    /// <remarks/>
    [System.Xml.Serialization.XmlElementAttribute("group")]
    public userTypeGroup[] group;

    /// <remarks/>
    public int uid;

    /// <remarks/>
    [System.Xml.Serialization.XmlIgnoreAttribute()]
    public bool uidSpecified;

    /// <remarks/>
    public string shell;

    /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(DataType="base64Binary")
]
    public System.Byte[] password;

    /// <remarks/>
    public string home_dir;

    /// <remarks/>
    public string name;

    /// <remarks/>
    public string comment;

    // ...
}
```

Note that the uid and uidSpecified class members form a pair. The value of the uidSpecified member indicates whether the uid is present or not in the data, meaning if it contains a meaningful value or not.

Before you try to read the value of the uid member, you have to check whether the element was included in the response. You do that by looking at the corresponding boolean flag first, i.e. the value of the uidSpecified member. If the value is true then uid was included in the response and contains a meaningful value. If the value is false then uid must be ignored.

When you assign a value to an "optional" member, you will have to set the corresponding boolean flag to true in order for the element to be included in the packet. Here's an example:

```
uid = 100;
uidSpecified = true;
```

If you don't set the "specified" flag to true then the receiving code will evaluate the optional element as "not included in the request".

Complex Types

The XML schema's complex types are represented in C# by strings, classes, and structures. An example of such an element is the initial_group member from the code example above. To determine whether the element is present or not in the packet, check if the value of the object is null:

```
if (initial_group == null)
{
    // The element is absent ...
}
```

Arrays

Finally, arrays (e.g. the group member in the example above) are considered optional if they have a null value or are empty.

Elements with No Content

Some of the elements in VZAgent protocol are used as flags. These are simple elements that have no type and never contain any data. In XML, you either include the element in a packet or you don't. For example, a schema may contain an element default. If you include it in the request as an empty element like this <default/>, you instruct the receiving code to use some default value.

In XML you simply include or omit the element in a packet. In C#, when passing an object of this kind to a method, you have to create it as an empty object like this:

```
myObj = new Object();
```

Base64-encoded Values

Because XML is text-based, not all ASCII characters are allowed to be passed as plain text. That's why some elements of the VZAgent protocol are of base64-encoded type. In C#, elements of this kind are represented as byte arrays. You don't have to additionally encode the data meant for these arrays, just fill them with the necessary content. Here is an example:

```
VZA.login loginCred = new VZA.login();
System.Text.Encoding ascii = System.Text.Encoding.ASCII;
loginCred.user = user;
loginCred.password = ascii.GetBytes(password);
```

Polymorphism

Substitution groups

We use schema extension and substitution groups to implement subtype polymorphism in our XML schema. For example, we have an abstract type packageType that describes a generic software package that can be installed into an Environment:

```
<xs:complexType name="packageType">
  <xs:sequence>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="summary" type="xs:string">
    <xs:element name="platform" type="xs:string">
    <xs:element name="description" type="xs:string">
    <xs:element name="arch" type="xs:string">
  </xs:sequence>
</xs:complexType>
```

Another type called package_rpmType extends packageType adding elements that describe an RPM package (Linux software package).

```
<xs:complexType name="package_rpmType">
  <xs:complexContent>
    <xs:extension base="vzlt:packageType">
      <xs:sequence>
        <xs:element name="version" type="xs:string">
        <xs:element name="path" type="xs:base64Binary">
        <xs:element name="distributive" type="xs:string">
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

We then create a substitution group so that an API call accepting the base type would also accept the subtype.

```
<xs:element name="package" type="vzlt:packageType"/>
<xs:element name="package_rpm" type="package_rpmType"
substitutionGroup="package"/>
```

.NET will generate a base class for the base type and subclasses of that base class for all the subtypes. You don't have to do anything in particular in order to use these classes. Just be aware that extended types translate into class hierarchies in C# so the calls that use substitution groups will work fine in C# too.

Choice indicator

VZAgent XML schema uses the <choice> indicator to implement type overloading. The <choice> indicator specifies that either one child element or another can occur:

```
<xs:complexType name="credType">
  <xs:sequence>
    <xs:choice>
      <xs:element name="user" type="xs:string">
      <xs:element name="uid" type="xs:long">
    </xs:choice>
    <xs:choice>
      <xs:element name="group" type="xs:string">
      <xs:element name="gid" type="xs:long">
```

```
    </xs:choice>
    <xs:element name="umask" type="xs:int">
  </xs:sequence>
</xs:complexType>
```

In the schema fragment above, the first `<choice>` element contains the `<user>` and `<uid>` elements. This means that you can use either one but not both at the same time (the second `<choice>` element works in the same exact manner). .NET will generate the code that will look similar to the following example:

```
public class credType {

    /// <remarks/>
    [System.Xml.Serialization.XmlElementAttribute(
         "user", typeof(string))]
    [System.Xml.Serialization.XmlElementAttribute(
         "uid", typeof(long))]
    public object Item;

    /// <remarks/>
    [System.Xml.Serialization.XmlElementAttribute(
        "gid", typeof(long))]
    [System.Xml.Serialization.XmlElementAttribute(
        "group", typeof(string))]
    public object[] Items;

    /// <remarks/>
    public int umask;

    /// <remarks/>
    [System.Xml.Serialization.XmlIgnoreAttribute()]
    public bool umaskSpecified;
}
```

The `credType` class represents the complex type `<credType>` from the XML schema example (see above). The `Item` object is a representation of the first `<choice>` element. The Items object represents the second `<choice>` element. As you can see, the object contains the same elements as the original XSD: `user, uid, gid, group, umask`.

This is how you use the generated code in your program:

Create the `credType` object.

```
VZA.credType myCredentials = new VZA.credType();
```

Let's say that we want to use the `<uid>` element. Simply assign a value of the appropriate type (`long`, in this case) to the `Item` object.

```
myCredentials.Item = 500;
```

If we wanted to use the `<user>` element, we would do this instead:

```
myCredentials.Item = "johndoe";
```

The second `<choice>` element differs from the first one only in that it is an array in the C# code:

Using the `<group>` element.

```
myCredentials.Items[1] = "root";
```

Using the <guid> element.

```
myCredentials.Items[1] = 0;
```

Timeouts

.NET SP1 has the default timeout value for the XML Web service calls set to 100000 ms. If you use this default value, some of your calls will never have a chance to complete. We've experienced the following error message related to this problem:

```
An unhandled exception of type 'System.Net.WebException' occurred in
system.Web.services.dll Additional information: The operation has
timed-out.
```

You may receive a different message but the cause may still be the same -- the default timeout value is too low. To avoid this problem, set the appropriate timeout value or set the timeout value to infinite, as shown in the following example:

```
MyService service1 = new MyService();

// Infinite timeout.
service1.Timeout = -1;

// The timeout is set to 10 minutes.
service1.Timeout = 10 * 60 * 1000;
```

Get/Set Method Name Conflict

Problem:

Microsoft Visual C# .NET may produce errors when generating the client code from WSDL similar to the following example:

```
<xs:element name="set_xxx">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="xxx" type="XXXtype" />
     </xs:sequence>
   </xs:complexType>
 </xs:element>
```

Note that the function `set_xxx` has a parameter `xxx`. Microsoft Visual C# .NET will generate the following code:

```
public partial class set_xxx {
    private string xxxField;
    /// <remarks/>
    public string xxx {
        get {
            return this.xxxField;
        }
        set {
            this.xxxField = value;
        }
    }
}
```

As you can see, the function has the same name as the class name. This causes the C# compiler to produce the following error:

```
error CS0542: 'set_xxx': member names cannot be the
               same as their enclosing type
```

Solution:

Create a batch file `wsdlc.bat` containing the following code and save it in your project directory:

```
setlocal
set WS=%1Web References\VZA
copy "%WS%\Reference.map" "%WS%\Reference.discomap"

REM The following block of code should be placed on
REM the same line. Every segment that starts
REM on the next line here, must be placed after
REM the previous one separated by a single space.

"%VS80COMNTOOLS%\..\..\SDK\v2.0\Bin\wsdl.exe"
    /l:CS /fields
        /out:"%WS%\Reference.cs"
           /n:%2.VZA "%WS%\Reference.discomap"

REM End of "The following block of code ..."

del "%WS%\Reference.discomap"
endlocal
```

```
exit /b 0
```

The file generates the new `Reference.cs` file (the file containing the proxy classes) fixing the problem described above by generating the regular properties instead of C#-style get/set fields. Do not try to run the file. It will by run automatically after we complete the rest of the steps.

In the Microsoft Visual C# .NET development environment, select **Project** > **Properties** menu item. Select **Build Events** option in the left pane. Now in the right pane, modify the parameter **Pre-build Event Command Line** to contain the following line:

```
$(ProjectDir)wsdlc.bat $(ProjectDir) $(ProjectName)
```

Note: Make sure that the `Reference.cs` file is not currently opened in the IDE, otherwise the compiler will use it instead of the new file that will be generated by our batch file.

Select the **Build** > **Build Solution** menu option to build your solution. This will take longer than usual because the `wsdlc.bat` file that we created will re-generate the proxy classes.

After the build is completed, the `Reference.cs` file will contain the newly generated stubs. At this point you can remove or comment out the entry that used in the **Project** > **Properties** > **Pre-build Event Command Line** option. If you do not, the stubs will be re-generated every time you build your solution.

If you decide to update the client code from WSDL located on our Web server again, make sure that you repeat the steps described here again.

The request describing this defect was submitted to Microsoft: `#FDBK46565`

Other SOAP Clients and Their Known Issues

Visual Basic .NET

.NET WSDL and XML parsers still have many bugs. Some of them prevent seamless usage of classes generated from VZA.wsdl.

After you add and try to compile the Web Reference from `http://www.swsoft.com/webservices/vza/4.0.0/VZA.wsdl`, you'll see the following compilation errors:

- `Keyword does not name a type.`
- `Reference to a non-shared member requires an object reference.`

The first error is caused by name conflicts between the user-defined identifiers and VB keywords. Usually parsers enclose the identifiers that are identical to VB keywords in square brackets. Note, however, that this does not work for words like `new`, which are encountered in WSDL and XSDs.

In our case, there are problems with the `get`, `stop`, `set`, and `select` function names. To solve them, simply double click on each error line in the Task list and enclose the respective words in square brackets.

The second error is related to the case-insensitive nature of VB -- it confuses the `system` field name in the VZAgent `cpu_loadType` class with its own `System` module. To fix this problem, change the line

```
<System.Xml.Serialization.XmlIgnoreAttribute()> _
```

to

```
<Xml.Serialization.XmlIgnoreAttribute()> _
```

Now you should have the code that compiles and works.

The first group of these problems does not exist in the Visual Studio 2005, but you still have to delete `System` from `Xml.Serialization.XmlIgnoreAttribute()` manually.

Visual J# .NET

Unfortunately, the current implementation of Visual J# in Visual Studio .NET 2003, due to its internal bugs, doesn't work with our WSDL. However, it works seamlessly with the Visual Studio 2005.

Apache Axis 1.2 for Java

For this client, we have one tip that reveals hidden knowledge of how to work with certificates:

```
System.setProperty("org.apache.axis.components.net.SecureSocketFactory
",org.apache.axis.components.net.SunFakeTrustSocketFactory");
```

The code above uses a fake trust manager trusting all certificates. This Java SOAP client also worked for us.

Using SOAP API

We're going to write our program in C# using the Visual Studio .NET IDE.

Managing Virtual Environments

Getting a List of Environments

Let's begin using VZAgent services and accessing Virtuozzo. As a starter we'll fetch the list of the VEs that exist on the server. This can be done with the "list_ve" command of the VZAgent HWM Operator. In SOAP-generated classes, the "list_ve" command corresponds to "hwmBinding.list_ve".

We also need a helper function to initialize bindings, set up the URL and the session ID:

```
    // Initialize new binding with Url and session ID.
    private System.Object InitBinding(System.Type bindingType)
    {
      System.Object Binding =
bindingType.GetConstructor(System.Type.EmptyTypes).Invoke(null);

      bindingType.GetProperty("Url").SetValue(Binding, m_url, null);
      VZA.packet_headerType header = new VZA.packet_headerType();
      header.session = m_sid;
      bindingType.GetField("packet_header").SetValue(Binding, header);
      return Binding;
    }
```

Once we have this helper function, we can write the code for fetching:

```
    // Fetch VPS list
    private string FetchVPSList()
    {
      // Initialize hwmBinding reflecting HWM Operator of VZAgent
      VZA.hwmBinding hwm =
(VZA.hwmBinding)InitBinding(typeof(VZA.hwmBinding));

      // Retrieving list of VPSes for an empty list of VEID, which
means to retrieve the whole list.
      VZA.list_ve lve = new VZA.list_ve();
      long[] lstve = hwm.list_ve(lve);

      // Serializing the resulting list into a string.
      String resp = "";
      for (int i = 0; i < lstve.Length; ++i)
      {
        resp += "VEID:" + lstve[i] + "\n";
      }

      return resp;
    }
```

To invoke FetchVPSList(), we need to add it to the very end of Run(), our main function:

```
      // Fetch and write VPS list.
      Console.WriteLine("VPS list:");
      Console.Write(FetchVPSList());
```

Finally, we start fetching the VPSs:

```
Session ID: vzagent0.30000.1dc41b999eat449bc702
VPS list:
VEID:1
VEID:101
```

```
VEID:102
VEID:103
VEID:104
VEID:105
VEID:200
VEID:301
VEID:444
VEID:517
VEID:520
```

VEID:777

Performance Monitor

Performance Monitor is an operator that allows to monitor the utilization of operating system resources. The operator reports detailed data about the resources used by specific components of the operating system on a periodic basis. Performance Monitor obtains the resource utilization data from the periodic collectors. Periodic collectors run at all times collecting the resources utilization information at predefined time intervals (60 minutes by default) and logging it to a history database.

Note: Due to existing SOAP clients limitations, the performance monitoring is not fully supported in the current version of our SOAP API. Specifically, you cannot use the monitor to receive notifications. You can still use the monitor to receive performance data reports but only on an on-demand basis, i.e. you make a request and you receive a single report containing the latest available data.

The rest of this section describes how to use Performance Monitor in your client programs.

Teminology

Before we can begin using Performance Monitor, we have to discuss its terminology.

Performance class

A logical group of monitored parameters. For example network throughput, CPU performance, memory usage, etc. You obtain the names of classes supported by a given operating system by retrieving them from VZAgent vocabulary.

Performance counter

A single parameter inside a performance class. Each counter is related to a specific area of system functionality and contains the actual performance data. For example, the number of sent and received packets for network class, used and free memory or swap space for memory class, etc.

Instance

In certain cases, there may be several objects in the system, a performance class is applicable to. For example, several network interfaces, several partitions, etc. These objects are called class instances, or simply instances. Some objects can have only one instance, for example memory (there can be only one memory) or swap.

Counter value type

The values that the counters contain can be either integers or floats. There are two parameters in a vocabulary describing it: `<value_type>` and `<type>`. The `<value_type>` parameter is used by Performance Monitor to correctly parse data and in the C++ calls. It is an equivalent of boolean `isInt`, i.e. 1 means integer value and 0 means a floating point. The `<type>` parameter is a string representation of counter type and can contain either "int" or "float" string values.

Counter type

There are two types of counters: incremental and absolute.

Incremental counters represent the total resource utilization over a period of time. For incremental counters, the next value is always greater than or equals to the previous value. A good example is a network counter that counts the total number of bytes the interface has sent or received since the Environment was started.

Absolute counters report the resource utilization at the precise moment. They provide performance snapshots that can later be compared and used to build the graph representation of the data, for example. These counters also provide minimum, maximum, and average values that are calculated by comparing the results of the two adjacent snapshots. The current value is the value of the counter at the moment the report is generated.

In the vocabulary, the `<counter_type>` element is used to specify the type of the counter. The two possible values that the element may contain are `0` for absolute counters and `1` for incremental counters.

Counters description in a vocabulary

Each performance class is represented by a category in vocabulary. To distinguish the class categories from the other categories, they all belong to another category named `counters`. In other words, each category that belongs to the `counters` category describes are performance class. The following is an example of a class description in a vocabulary:

```
<category>
  <id>counters_net</id>
  <category>generic</category>
  <category>counters</category>
  <short>Network usage</short>
  <long>Network usage related parameters</long>
</category>
```

The `<id>` element specifies the class name.

The `<category>` elements specify the categories that this class belongs to. A class can belong to more than one category. The `counters` category indicates that this is a performance class. The `generic` category specifies the Environment type to which this class applies. This is due to the fact that different Environment types may have same performance classes, but different sets of counters. The additional category element is used to distinguish the classes of different Environment types.

The following example illustrates how a counter that belongs to the `counters_net` category is described in a vocabulary:

```
<parameter>
  <id>counter_net_incoming_bytes</id>
  <category>counters_net</category>
  <type>int</type>
  <value_type>1</value_type>
  <counter_type>1</counter_type>
  <short>Incoming traffic (bytes)</short>
  <long>Amount of incoming network traffic in bytes</long>
  <measure>bytes</measure>
</parameter>
```

The `<id>` element specifies the counter name.

The `<category>` element specifies the category (the performance class) that the counter belongs to.

The <type> element specifies the counter data type. In this case, the value is `int`, meaning that the values that the counter reports are integers.

The `<counter_type>` element specifies the counter type. In this instance, the type `1` indicates that this is an incremental counter.

The `<measure>` element specifies the units of measure for this counter.

Retrieving Classes and Counters From Vocabulary

The following example illustrates how to retrieve all performance classes and their counters from the vocabulary.

Using Performance Monitor

Now that we know what classes, instances, and counters are, we a ready to use Performance Monitor. The first thing that you have to do is select the resource that you would like to monitor. You do that by selecting the name of the class and the names of the counters from the vocabulary. Let's say that we want to monitor the system CPU usage. From the example provided in the previous section, we know that the name of the counter that we need is `counter_cpu_system`. The following call retrieves the latest available data for the counter:

<code example>

The `<eid>` parameter contains the ID of the Environment to monitor (in our example, the ID of the Environment we are currently connected to). We are using just one counter from the `counters_cpu` class in our call. If you wish, you can use all counters from this class. Simply remove the `<instance>` element together with the `<counter>` element. You can also add other classes and their counters and even get the data for multiple environments (more on that later).

The following is an example of a performance data report:

<code example>

As you can see, the report contains the current, average, minimum, and maximum values. Since this is an incremental counter (see previous section), the current value is the total system CPU time (the time spent by the processor to execute operating system tasks), and the average, minimum, and maximum values all contain the difference (the increase) between the current value and the value from the last report.

Getting the Data for Multiple Environments

Performance monitor allows to retrieve the data for multiple environments in one call. If you are connected to a server that hosts virtual servers, not only you can get the data for the host server itself but you can also get the data for the virtual servers as well.

Let's say that we want to retrieve the performance data for the host server and each Virtuozzo VE that it hosts. In order to do that, we would have to retrieve the names of the classes and the performance counters for each class. The question is, since we are going to be monitoring two different types of environments (generic and Virtuozzo), we have to retrieve the classes and the counters for each Environment type individually. The problem is, there's no guarantee that both types of environments will contain the same classes and counters. So, if you want to monitor multiple environments of different types, you have to make sure that you use a class and the counters that are available for both Environment types.

If you specify just the class name and omit the counters in the request, the monitor will retrieve the counters automatically and will display the "correct" counters for each Environment type, i.e the report will show different counters for different Environment types.

Internal vs. External Monitoring

When it comes to virtual environments, there are two possible types of resources utilization information that Performance Monitor can acquire for you. Let's say that we have a server hosting virtual environments (Virtuozzo VEs for example). We can monitor the resources utilization by the host itself or we may see the impact that a particular virtual Environment makes on the host server. In other words, if we start a monitor on the host server and specify the ID of a virtual Environment to monitor, we will see how the virtual Environment uses the resources of the host server, not its own resources. If you want to see the utilization of operating system resources inside the virtual Environment, you will have to install VZAgent on it, connect to the Environment directly, and start a monitor there.

Setting Up a Cluster

We already discussed the VZAgent cluster early in this guide. In this section, we'll talk about how to set it up. The following steps describe the cluster setup procedure:

Step 1

First, you have to install VZAgent on every machine that will participate in a cluster.

Step 2

Then you have to decide which machine you want to be the master Environment in the cluster. There aren't any specific requirements regarding the type of the machine or the software that it's running (other than VZAgent software). Keep in mind, however, that the load on the master Environment will be significant, especially considering all the network communications between the master Environment and all its slave environments. You should choose the machine with enough processing power and high-bandwidth network capabilities. This step doesn't require a programming or any other effort. Simply choose the machine that you want to be the master.

Step 3

The cluster is formed by adding the slave environments to the master Environment. In order to do that, connect to VZAgent on the master Environment and execute the following call:

The `<con_info>` structure in the call above contains the connection information that the master Environment will use to connect to the Environment being added as a slave (substitute the values with those appropriate to your server). As a result of this call, the master Environment will connect to the target Environment and will register it as a slave Environment. The master Environment will make the appropriate configuration changes that will identify it as a master. The slave Environment will also make the necessary configuration changes that will identify it as a slave managed by this master Environment.

You now have a cluster. To add more slave environments to the cluster, repeat Step 3 for every one of them. You can add as many environments to the cluster as you wish.

To display the complete list of environments available in the cluster, make the following call:

The response message will contain the list of environments:

If you want to remove an Environment from a cluster (for example, you want to add it to another cluster), execute the following call supplying the Environment ID:

If you want to destroy the cluster altogether, delete all the slave environments from it first and then execute the following call:

The call will remove all the configuration properties from the master Environment identifying it as a master, thus destroying the cluster. Without this step, you wouldn't be able to add a former master Environment to another cluster as a slave.

Troubleshoting

I'm receiving one of the following errors when trying to connect to the server:

```
The underlying connection was closed: Unable to connect to the remote
server.
```

Solution: Check your URL, port and routing to your server.

```
http://schemas.xmlsoap.org/soap/envelope/:Server, VZAgent responded
with error
Details:
2704
Authentication failure - either user name or password is incorrect
```

Solution: Check your login and password.

Somewhere in the middle of an operation, I get the following error:

```
http://schemas.xmlsoap.org/soap/envelope/:Server, VZAgent responded
with error
Details:
1004
Error invoking vzctl utility: VE is already running
```

Solution: Check the state of your VE that is used for the current operation. In the example above, you try to start a VE and get the error message. This kind of error may occur when you are starting a VE that is already in the "running" state.

I'm using SOAP with .NET Web Services and I get the following error:

```
An unhandled exception of type 'System.Net.WebException' occurred in
system.Web.services.dll Additional information: The operation has
timed-out.
```

Solution: .NET SP1 has the default timeout value for the XML Web service calls set to 100000 ms. To avoid this problem, set the appropriate timeout value or set the timeout value to infinite, as shown in the following example:

```
MyService service1 = new MyService();

// Infinite timeout.
service1.Timeout = -1;

// The timeout is set to 10 minutes.
service1.Timeout = 10 * 60 * 1000;
```

Microsoft Visual C# .NET 2005 does not compile SOAP applications in Release mode.

When attempting to perform a Release build, the `sgen.exe` throws `Out Of Memory` exceptions.

This is a known defect in Microsoft **Sgen** tool. To fix this problem, try setting the option **Project** > **Properties** > **Build** > **Generate serialization assembly** to `Off` to avoid calling `sgen.exe`.

Advanced Topics

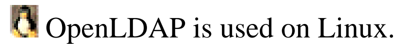
Authentication and Authorization

Authentication refers to a process of verifying the identity of a user.

Authorization refers to a process of establishing the user access rights.

Directory as User Database

VZAgent uses Lightweight Directory Access Protocol (LDAP) and LDAP-compliant directory services for user authentication and management. Depending on the platform, the following directory services are used:


OpenLDAP is used on Linux.

Active Directory Application Mode (ADAM) is used on Windows.

LDAP directory is object-oriented. Similar to other object-oriented models, LDAP uses a concept of classes and objects. Each entry in a directory is an instance of a class, or an *object*. The objects are organized into a tree. Bottom-level objects may be joined to form higher-level objects. Objects that contain other objects are called *containers*. Each directory has a single root object -- a container called *domain component*. All other objects, including other containers, are stored within the domain component container.

Each class has attributes to describe an object and to meet the object storage requirements. The definitions of classes and attributes form a directory *schema.* A default schema has a set of predefined classes and attributes but can be extended with additional classes and attributes if needed.

Users and groups are the bottom level objects in a directory hierarchy. Every entry in an LDAP directory has a *distinguished name*, which uniquely identifies the entry in the directory. A distinguished name (or DN for short) is composed of the names of all levels of the hierarchy from bottom to top. For example, let's say that the user `admin` belongs to the container `Users`, which belongs to domain component `Mydomain`. The distinguished name for the user `admin` will look like this:

```
CN=admin,CN=Users,DC=Mydomain
```

A DN is similar to a fully-qualified file name in a file system -- it is a unique name for an entry in a directory tree. Because distinguished names are not very user-friendly, VZAgent allows using plain user and group names in its API calls through *default distinguished names*.

While users and groups are stored as objects in the directory tree, the data associated with a user is stored using object attributes. As we said earlier, classes and attributes constitute a directory schema. If the default schema does not satisfy your storage requirements, you may extend it by adding your own attributes. Both the directory structure and the schema can be managed with the standard directory management tools. Users and groups can also be managed through VZAgent API. In addition, a low-level directory management interface is also included in the VZAgent API.

Realms

VZAgent is capable of working with multiple directories residing on the same or different machines. A directory may be installed in the local Environment, in the master Environment, or anywhere on the network. VZAgent uses a special construct called a *realm* to describe a directory in terms of connectivity. Realms are defined in the VZAgent configuration. Each realm has a name, and consists of the directory connection parameters. During authentication, a user may specify which directory he/she wants to be authenticated against by supplying the name of the realm containing the directory connection information. VZAgent will try to find the specified realm in its configuration and, if the realm is found, will use the values stored in it to connect to the directory and to authenticate a user against it. The VZAgent configuration has one default realm. When VZAgent is installed, the default realm refers to the directory installed in the local Environment. By not supplying a realm information during login procedure, a user instructs VZAgent to use the default realm.

One of the main advantages of using realms is the ability to use external directories. For example, if you have a directory somewhere on the network that already contains all your users, you don't have to move them to the local directories on individual machines. Instead, you may define a realm for the directory in each participating Environment by specifying the directory connection parameters and then use that realm for user authentication. Another benefit is the ability to use multiple directory instances on the same server.

The following is a list of the parameters that define a realm:

Parameter	Description
Address	The IP address or a hostname of the machine where the directory is located.
Port	The TCP port number used by the directory instance. Every directory instance is listening on a certain port, which is assigned to it when the directory is created. If you have multiple instances running on the same machine, each instance will be using its own port.
Default	A boolean value specifying whether this is a default realm (the default realm is used when no realm ID is specified in a call).
ID	The unique realm identifier (automatically generated).
Name	Realm name.
Base DN	The distinguished name of the root object in the directory.
Default DN	The distinguished name of the default container in the directory tree used for storing user information.
Login	Login information to connect to a directory instance.

Sessions

VZAgent provides *sessions* as a mechanism for applications to authenticate a user. After the identity of a user has been verified, a logon session is created for the user on the server side and the unique session ID is passed back to the client application. The application then includes the session ID in every request made in the user's name in order for VZAgent to recognizes the user who owns the session, and to make a decision whether the call should be authorized or not. The session mechanism is used automatically for permanent connections (the connection established during the original login). It is possible to establish additional, temporary connections with VZAgent. With the temporary connections, it is the responsibility of the programmer to pass the session ID to VZAgent in order for it to recognize the user.

VZAgent Configuration

VZAgent configuration consists of a set of configuration parameters for each of the VZAgent operators. The configuration information is stored in a file as an XML document. On VZAgent startup, a corresponding director reads the information from the configuration file and uses it to configure the operators. As a result, all operators are initialized with the parameters currently stored in the configuration file.

Because the configuration information is stored as an XML document, it can be edited and sent to VZAgent from a client program just like any other request. VZAgent, receiving the configuration data, will create a new configuration file replacing the existing file. At the same time all free-pool operators are released, the busy operators are marked for exiting on message processing completion, and the new operators are invoked newly configured. The single operators -- the operators that have no pool and are running at all times -- handle the configuration message as a regular request and reconfigure themselves on the fly. VZAgent configuration information can be retrieved and modified using VZAgent API.

Vocabulary

A significant part of the VZAgent knowledge base is placed into a vocabulary. This is done in order to provide the client software with a persistent information, which would otherwise have to be hard-coded. The vocabulary contains a vast set of categorized parameters with long and short descriptions, maximal, minimal, and default values, width and alignment, types and measures, etc. Any client program working with VZAgent should read the vocabulary as soon as possible in order to speak the same language as the operating system on the server that the program is accessing through VZAgent. The vocabulary contains information about QoS counters with their names and descriptions, about services and the possible actions on them, about predefined network protocols and available configuration files, about processes information parameters and QoS validation formulas.

VZAgent Group

This section describes the VZAgent Group architecture and provides examples of how the Environments registered in a group can interact with each other.

Virtuozzo Group Architecture

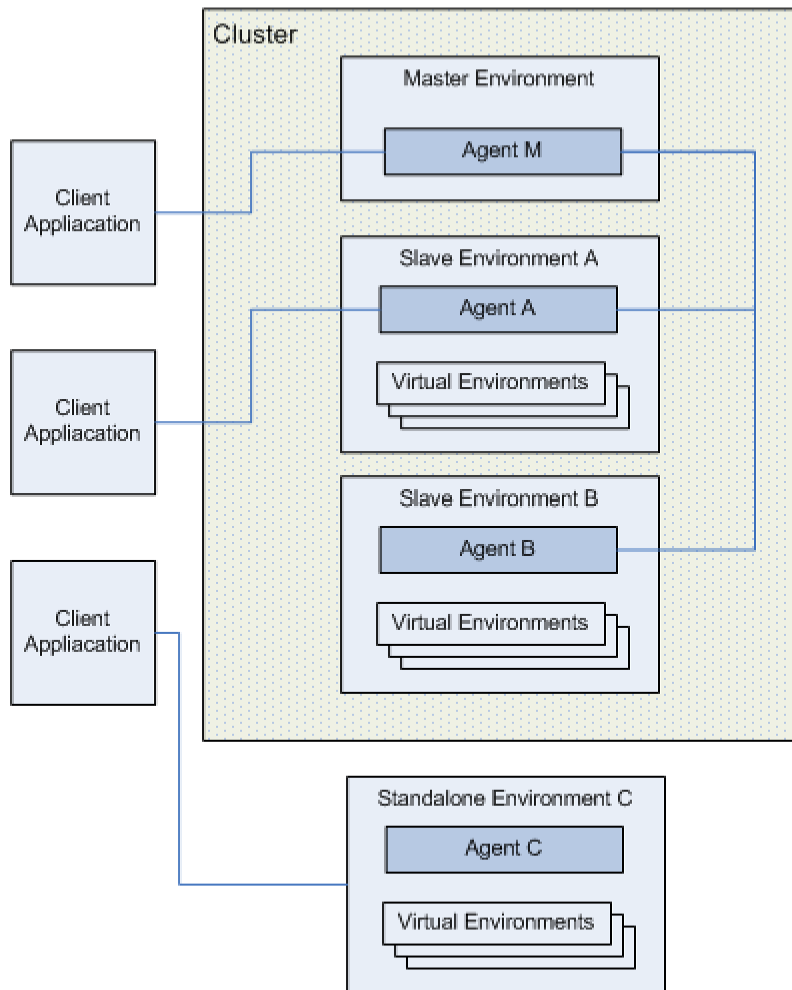


Figure 2: Cluster Architecture

The term Virtuozzo group refers to a network of Environments each running its own VZAgent software and interconnected with each other by means of internal VZAgent mechanisms. The central component of a Virtuozzo group is the master Environment. The master Environment administers the entire group by allocating, monitoring, and controlling the group resources. By default, any standalone Environment is a master Environment to itself, which means that any Virtuozzo group functionality can be used even on a standalone Environment. In a multiple Environment scenario, there's one master Environment and many slave Environments. All immediate child Environments (the Environments A and B on the diagram above) must be registered with the master Environment. As soon as this is done, VZAgent in the master Environment begins monitoring the slave Environments and can access any of them, including all virtual Environments hosted by them. A client application connected to VZAgent in a master Environment can access any Environment in a group, including the slave Environments and all of the virtual Environments that they host.

The Environments in a group are organized in a hierarchical structure. The master Environment is the root node of the entire group tree. Slave Environments form the second level of the hierarchy. Virtual Environments hosted by the slave Environments form the third level. If the virtual Environments in the third level host any virtual Environments of their own, then those Environments will make up the next level, and so forth. The following diagram demonstrates the Environment nesting within a single slave or a standalone Environment.

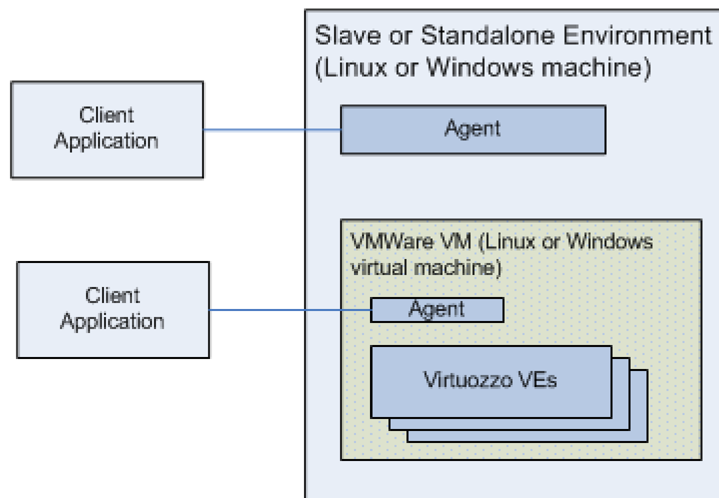


Figure 3: Nested Environments

In the diagram above, a slave Environment (or a standalone Environment) has VZAgent running on it. The Environment hosts a VMware Virtual Machine, which also has VZAgent running in it. In addition, the VMware VM hosts a number of Virtuozzo VEs. A client program connected to VZAgent running in the slave Environment can manage the Environment itself, VMware VM, and any of the Virtuozzo VEs. A client program connected to VZAgent running in the VMware VM can access only the VM and the Virtuozzo VEs that the VM hosts. The slave Environment is said to be a parent Environment of the VMware VM. In turn, the VMware VM is a parent Environment to each Virtuozzo VE that it hosts.

Scheduler

In order to be able to process more requests and decrease the load on the host Environment, VZAgent features a simple internal scheduler.

Message Classification and Priorities

The messages traveling through VZAgent are divided into four categories (classes), according to their priorities and processing time. Before discussing these categories, it is necessary to mention that the priorities of user messages (priorities of the messages coming from the clients to VZAgent before their processing by the operator connection) are not the same as the priorities of the internal VZAgent messages. The former are translated into the latter by operator connection, i.e. on their entrance to VZAgent.

The four categories (classes) of the messages are:

- Normal messages (default).
- Urgent messages.
- Heavy messages.
- Emergency messages.

Normal messages take a moderate time to be processed (up to 5 minutes by default) and their priority ranges from -999 to +999 to be set by the client and from -1999 to +1999 internally. They may include such operations as stopping an Environment, getting services states, and the like.

Urgent messages take very little time to process (no more than a minute) and have the client priority range from -3000 to -1000 and the internal priority range from -6000 to -2000. They may include getting a list of environments, retrieving a user information, starting a log, etc.

Heavy messages take significant time to be processed (from 5 minutes to hours or more). Their priority ranges from 1000 to 3000 if set by the client and from 2000 to 6000 internally. Among such messages are creating new virtual environments, cloning, migration, template installation, and others.

Emergency messages are for internal VZAgent use only and consequently have just internal priorities below -6000.

Internal messages also differ by the credentials of the original user sending them. Root messages have higher internal priorities than those issued by a regular user.

The table below summarizes the above considerations:

Messages	Heavy	Normal	Urgent
External priorities	1000 to 3000	-999 to 999	-3000 to 1000
Root internal priorities	2000 to 4000	-1999 to -1	-6000 to -4000
User internal priorities	4000 to 6000	1 to 1999	-4000 to -2000

Pool and Single Operators

For on-demand requests, the director provides pools of operators that are being forked and cached as necessary. For example, the Environment management operator can simultaneously serve up to 4 virtual Environment creation processes, up to 10 Environment stops, and up to 20 Environment configuration fetches by default. It means that the director forks another Environment management operator if all of the existing operators are busy. This works up to a certain limit, after which the next incoming message is queued and waits until one of the existing operators becomes available upon the completion of a previous request.

Pools are strictly concerned with a particular operators and don't intersect in any way. As an example, the computer management operator pool never interferes with the Environment management pool.

Any pool consists of two sets of operators. One set is comprised of the busy operators and the other contains the operators that are currently available. A new incoming message is sent to one of the available operators. The status of the operator is immediately switched from the "available" to "busy". Upon completion, the operator is put back to the "available" set unless the total number of operators in the pool has already reached the pool limit, in which case the operator instance is destroyed.

Static limits of pools (limits that are not changed with time unless VZAgent is reconfigured) consist of three values - one for each message class. The "heavy" limit allows no more heavy messages to be simultaneously run than the number represented by its value. The "total" limit does the same thing for normal plus heavy messages. And the "urgent" limit restricts the number of urgent + normal + heavy messages. Emergency messages are not limited. All this means that messages of all types are considered together and if a pool is partially busy with heavy messages, the number of normal messages to run is reduced, too. Operators for urgent messages are invoked even if the total limit is reached - that will only make the pool shrink back after the completion of any requests to a value not greater than the total limit.

The other type of operators are the *single operators*. These operators run at all times. Unlike the pool operators, they never fork additional processes. This type of operators include the periodic collectors (the operators that collect the data on a periodic basis), the event reporters (the operators that notify the client of the important system events), and some others.

Dynamic Limits

In an attempt to provide scalability depending on the system load, pool limits are dynamically changed. Their increase and decrease depend on the completion of processing a request. If the request is killed by the timeout, the system is considered to be too loaded for this many operations of the kind to be performed in parallel, and so the dynamic pool limit is decreased by 1 down to the minimum of 1. If the operation was successful, the dynamic pool limit is increased by the `1/comeback_ratio` value up to the corresponding static limit. It allows a faster reaction to heavy load peaks and slower recover. Dynamic limits exist for each of the static limits: normal, heavy, and urgent. Decreasing a dynamic limit happens not only for the limits of the given message class (judging by the message whose processing was terminated for the timeout), but also for heavier classes of messages. It means that the timeout of an urgent message will lead to all of the three dynamic limits being decremented. Incrementing a dynamic limit would also affects all the limits of lighter classes. Thus, the completion of a heavy message allows to increment the dynamic limits for all of the message classes. The incremental values are proportional to the corresponding static pool limits. Here is an example.

Suppose we have the following pools: 4 for heavy, 10 for normal and 20 for urgent messages and the `comeback_ratio` equalling 4. A successful completion of an urgent messages will result in the following increases.

Urgent Dynamic Limit = +1/comeback_ratio.

Normal Dynamic Limit = +1/comeback_ratio/(20/10).

Heavy Dynamic Limit = +1/comeback_ratio/(20/4)

This allows heavier limits not to stick near their minimums if messages of their class are not coming.

Queue

If a particular pool limit has been reached and the message at hand cannot be served immediately, it is placed in the queue. The queue is a priority-based collection. Higher-priority messages are placed before the lower ones. So, a queue overflow may lead to dropping some of the already queued messages that have a lower priority than those coming now. With the corresponding operators becoming available, the messages are unqueued and sent for processing.

The queue has the same principles not only for on-demand operators with their pools, but everywhere else (even for internal VZAgent messages).

Timeouts

Timeouts are set for every operation performed by the VZAgent on-demand operators. They are necessary for preventing system hangs and overloading. Different timeouts are set for each class of messages. This is achieved by introducing timeout limits.

A timeout limit is the maximal timeout value that can be set for a particular message class. By default, the timeouts are set at 5 minutes for normal messages, 1 minute for urgent messages, and 100 hours for heavy messages. All these values are configurable.

When the director receives an XML request message from a client, it sets the default timeout value for it by populating the `timeout_limit` attribute of the `packet` element (the root element of every message). This value specifies the maximum timeout allowed for this message class. Upon receiving the message, the operator verifies the specified timeout value. If it is satisfied with it, it proceeds with the processing of the request. If the value is greater than the timeout limit for the given message class, the operator returns the message to the director changing the value to the one it finds appropriate. The director then recalculates the priority of the message, places it into the corresponding message class, and reschedules it.

Index

A

About This Guide • 6
Advanced Sample Program • 44
Advanced Topics • 97
Apache Axis 1.2 for Java • 87
API • 10
Authentication and Authorization • 97
Authentication Concepts • 14
Authorization • 15

B

Base64-encoded Values • 80

C

Certificates Policy Preparation • 75
Configuring a Virtuozzo VE • 70
Connecting to VZAgent • 35
Connection URL • 75
Connectivity • 13
Creating a Simple Client • 73
Creating a Simple Client Application • 34
Creating and Configuring a Virtuozzo VE • 64
Creating the Virtual Environment • 69

D

Destroying a Virtuozzo VE • 70
Directory as User Database • 98
Documentation Conventions • 6
Dynamic Limits • 106

E

Elements with No Content • 79
Encoding and Decoding an XML message. • 45
Error Handling • 32
Events and Alerts • 70

F

Feedback • 8

G

General Conventions • 8
Generating Client Code from WSDL • 72
Get/Set Method Name Conflict • 84
Getting a List of Environments • 88
Getting a List of OS Templates • 67
Getting a List of Sample Configurations • 64
Getting the Data for Multiple Environments • 92

I

Installation • 10, 73
Internal vs. External Monitoring • 93
Introduction • 9, 71
Invoking Web Services • 77

K

Key Concepts • 12
Key Features • 72

L

Limitations • 72
Logging In • 36, 60
Logging in and Creating a Session • 76
Login and Session Management • 56

M

Managing Virtual Environments • 88
Message Body • 28
Message Classification and Priorities • 104
Message Header • 23
Multiple Calls and Targets • 30

N

Namespaces • 32

O

Optional Elements • 78
Organization of This Guide • 6
Other SOAP Clients and Their Known Issues • 86
Overview • 71

P

Performance Monitor • 89
Performance Monitoring • 70
Polymorphism • 81
Pool and Single Operators • 105
Populating the VE Configuration Structure • 68
Preface • 6

Q

Queue • 106

R

Realms • 15, 99
Restarting a Virtuozzo VE • 40
Retrieving a List of Virtuozzo VEs • 39
Retrieving a VE Configuration • 70
Retrieving Classes and Counters From Vocabulary • 91
Retrieving the Realm Information • 57

S

Scheduler • 103
Schema Tables • 20
Sessions • 62, 100
Setting Up a Cluster • 94
Shell Prompts in Command Examples • 7
SOAP Bindings • 77
Some Programming Techniques • 54
Starting, Stopping, Restarting VZAgent • 11
Step 1
Choosing a Development Project • 73
Step 2
Generating the Stubs from WSDL • 73
Step 3
Adding the Code • 74
Step 4
Running the Sample • 77
Summary • 41

T

Teminology • 90
Terminology • 15
The Complete Program Code • 42, 46
The Location of WSDL • 72
The Null-Terminating Character • 31
Timeouts • 83, 107
Troubleshoting • 96
Typographical Conventions • 7

U

Using Performance Monitor • 92
Using SOAP API • 71, 87
Using XML API • 18

V

Virtuozzo Group Architecture • 102
Visual Basic .NET • 86
Visual J# .NET • 86
Vocabulary • 100
VZAgent Architecture • 12
VZAgent Configuration • 100
VZAgent Group • 101
VZAgent Messages • 19
VZAgent Overview • 9

W

Who Should Read This Guide • 6

X

XML API Basics • 18
XML Libraries • 55
XML Schema • 18