

PVA Agent

Programmer's Guide

1.0

ISBN: N/A
Parallels Holdings, Ltd.
c/o Parallels Software, Inc.
13755 Sunrise Valley Drive
Suite 600
Herndon, VA 20171
USA
Tel: +1 (703) 815 5670
Fax: +1 (703) 815 5675

Copyright © 1999-2008 Parallels Holdings, Ltd. and its affiliates. All rights reserved.

Parallels, Coherence, Parallels Transporter, Parallels Compressor, Parallels Desktop, and Parallels Explorer are registered trademarks of Parallels Software International, Inc. Virtuozzo, Plesk, HSPcomplete, and corresponding logos are trademarks of Parallels Holdings, Ltd. The Parallels logo is a trademark of Parallels Holdings, Ltd.

This product is based on a technology that is the subject matter of a number of patent pending applications. Virtuozzo is a patented virtualization technology protected by U.S. patents 7,099,948; 7,076,633; 6,961,868 and having patents pending in the U.S.

Plesk and HSPcomplete are patented hosting technologies protected by U.S. patents 7,099,948; 7,076,633 and having patents pending in the U.S.

Distribution of this work or derivative of this work in any form is prohibited unless prior written permission is obtained from the copyright holder.

Apple, Bonjour, Finder, Mac, Macintosh, and Mac OS are trademarks of Apple Inc. Microsoft, Windows, Microsoft Windows, MS-DOS, Windows NT, Windows 95, Windows 98, Windows 2000, Windows XP, Windows 2003 Server, Windows Vista, Microsoft SQL Server, Microsoft Desktop Engine (MSDE), and Microsoft Management Console are trademarks or registered trademarks of Microsoft Corporation.

Linux is a registered trademark of Linus Torvalds.

Red Hat is a registered trademark of Red Hat Software, Inc.

SUSE is a registered trademark of Novell, Inc.

Solaris is a registered trademark of Sun Microsystems, Inc.

X Window System is a registered trademark of X Consortium, Inc.

UNIX is a registered trademark of The Open Group.

IBM DB2 is a registered trademark of International Business Machines Corp.

SSH and Secure Shell are trademarks of SSH Communications Security, Inc.

MegaRAID is a registered trademark of American Megatrends, Inc.

PowerEdge is a trademark of Dell Computer Corporation.

eComStation is a trademark of Serenity Systems International.

FreeBSD is a registered trademark of the FreeBSD Foundation.

Intel, Pentium, Celeron, and Intel Core are trademarks or registered trademarks of Intel Corporation.

OS/2 Warp is a registered trademark of International Business Machines Corporation.

VMware is a registered trademark of VMware, Inc.

All other marks and names mentioned herein may be trademarks of their respective owners.

Contents

Preface	6
About This Guide	6
Who Should Read This Guide	6
Organization of This Guide	7
Documentation Conventions.....	7
Typographical Conventions.....	8
Shell Prompts in Command Examples	9
General Conventions	9
Feedback	10
Getting Started	11
Parallels Agent Overview	11
Parallels Agent API.....	12
System Requirements	13
Installation	13
Starting, Stopping, Restarting.....	14
Location of XSD and WSDL.....	15
Agent Architecture.....	16
Connectivity.....	17
Authentication Concepts.....	18
Realms	18
Authorization.....	19
Terminology	19
Using XML API	21
XML API Basics.....	21
XML Schema	22
Agent Messages.....	22
Error Handling.....	34
Creating a Simple Client Application	35
Connecting to Agent.....	36
Logging In	38
Retrieving a List of Virtuozzo Containers.....	41
Restarting a Virtuozzo Container	42
Summary	43
The Complete Program Code	44
Login and Session Management	46
Retrieving Realm Information.....	47
Logging In	50
Sessions	52
Logging In To VZCC or VZPP	54
Creating and Configuring Virtuozzo Containers	57
Getting a List of Sample Configurations.....	58
Getting a List of OS Templates	60
Populating Container Configuration Structure	61
Creating a Virtuozzo Container	63
Retrieving Container Configuration	65
Configuring a Virtuozzo Container	66

Destroying a Virtuozzo Container	69
Performance Monitor	69
Classes, Instances, Counters	70
Getting a Performance Report	71
Receiving Periodic Reports	73
Monitoring Multiple Environments	74
Events and Alerts	75
Request Routing	78
Using SOAP API	81
Introduction	81
Overview	81
Key Features	82
Limitations	82
Generating Client Code from WSDL	82
Creating a Simple Client Application	82
Step 1: Choosing a Development Project	83
Step 2: Generating Proxy Classes From WSDL	83
Step 3: Main Program File	84
Step 4: Running the Sample	90
Complete Program Code	91
Developing Agent SOAP Clients	96
SOAP API Reference	96
Optional Elements	97
Elements with no Content	98
Base64-encoded Values	98
Timeouts	99
Get/Set Method Name Conflict	100
Managing Virtuozzo Containers	101
Creating a Container	102
Starting, Stopping, Restarting a Container	105
Destroying a Container	106
Suspending and Resuming a Container	107
Getting Container Configuration Information	108
Configuring a Container	109
Cloning a Virtuozzo Container	115
Migrating a Container to a Different Host	117
Backup Operations	120
Performance Monitor	135
Monitoring Alerts	139
Other SOAP Clients and Their Known Issues	142
Visual Basic .NET	142
Visual J# .NET	142
Apache Axis 1.2 for Java	143
Troubleshooting	144
Advanced Topics	145
Agent Configuration	145
Internal Request Scheduler	145
Message Classification and Priorities	146
Pool and Single Operators	147
Dynamic Limits	148
Queue	148
Timeouts	149

Appendix A: Performance Counters	150
----------------------------------	-----

Index	158
-------	-----

CHAPTER 1

Preface

In This Chapter

About This Guide.....	6
Who Should Read This Guide.....	6
Organization of This Guide.....	7
Documentation Conventions.....	7
Feedback	10

About This Guide

This guide describes how to develop client applications using Parallels Agent API. This documentation exists as HTML, HTML Help, and Adobe Acrobat documents. You can browse the HTML version of this document on the Web. A link to the document is available on the Documentation page on the SWsoft site.

Who Should Read This Guide

Primary audience for this guide is anyone developing Parallels Agent client applications. To use this book you should have UNIX or Windows system administration experience and a good knowledge of Virtuozzo Containers software. 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

This guide is organized into the following chapters:

Chapter 1, Preface. Provides information about this guide.

Chapter 2, Getting Started. Provides an overview of the Parallels Agent software. Includes installation instructions. Talks about Parallels Agent architecture. Lists communication protocols that can be used to communicate with Agent. Introduces user authentication concepts. Provides a list of the most commonly used Agent terms and explains their meaning.

Chapter 3, Using XML API. This chapter is intended for those who would like to develop client applications using XML API. It begins with the explanation of the XML API basics. It then provides a complete example of how to create a simple client application in Perl. The rest of the material describes how to perform the most common Parallels Virtuozzo Containers management tasks and provides XML code samples.

Chapter 4, Using SOAP API. This chapter is intended for the developers who would like to develop client applications using SOAP API. It begins with the SOAP API overview, its key features and limitations, and provides information on how to generate client code from WSDL documents. It then provides step-by-step instructions on how to create a simple client application in C#. It continues with the description of how to avoid certain problems and how to handle some of the programming issues. It then describes how to perform the most common tasks providing C# code samples. It also provides information about other SOAP clients and their known issues. A troubleshooting information is included at the end of the chapter.

Chapter 5, Advanced Topics. This section describes some of the advanced Agent features that you can use to fine-tune your client applications.

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.

The table below presents the existing formatting conventions.

Formatting convention	Type of Information	Example
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.

<i>Italics</i>	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 <i>EZ templates</i> . To destroy a Container, type <code>vzctl destroy <i>ctid</i></code> .
Monospace	The names of commands, files, and directories.	Use <code>vzctl start</code> to start a Container.
Preformatted	On-screen computer output in your command-line sessions; source code in XML, C++, or other programming languages.	Saved parameters for Container 101
Monospace Bold	What you type, as contrasted with on-screen computer output.	# rpm -V virtuoizzo-release
Key+Key	Key combinations for which the user must press and hold down one key and then press another.	Ctrl+P, Alt+F4

Besides the formatting conventions, you should also know about the document organization convention applied to Parallels documents: chapters in all guides are divided into sections, which, in turn, are subdivided into subsections. For example, **About This Guide** is a section, and **Documentation Conventions** is a subsection.

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.	<i>To create a Container:</i>
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.
<i>Italics</i>	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 <i>EZ templates</i> . To destroy a Container, type <code>vzctl destroy <i>ctid</i></code> .
Monospace	The names of commands, files, and directories.	Use <code>vzctl start</code> to start a Container.

Preformatted	On-screen computer output in your command-line sessions; source code in XML, C++, or other programming languages.	Saved parameters for Container 101
Monospace Bold	What you type, contrasted with on-screen computer output.	# rpm -V virtuoizzo-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!

The ideal place for your comments and suggestions is the Parallels documentation feedback page (<http://www.parallels.com/en/support/usersdoc/>).

CHAPTER 2

Getting Started

In This Chapter

Parallels Agent Overview	11
System Requirements	13
Agent Architecture	16
Connectivity	17
Authentication Concepts	18
Terminology	19

Parallels Agent Overview

Parallels Agent is a server-side software that enables the development of client applications that can manage and monitor Parallels Virtuozzo Containers. Virtuozzo Containers can be managed through standard tools that come with Parallels Virtuozzo Containers, including the command line and GUI tools. With Parallels Agent, you can build your own custom applications that communicate with Parallels Virtuozzo Containers directly. Using Parallels Agent APIs, you can programmatically integrate Parallels Virtuozzo Containers with external software products or to build your own management and monitoring tools.

The following list describes the most common operations that can be performed through Parallels Agent:

- Create and destroy a Virtuozzo Container.
- Start, stop, restart a Container.
- Migrate, clone, move a Container to a different location.
- Create Container backups.
- Get Container status and configuration information.
- Modify Container configuration parameters.
- Obtain current statistical data and resource usage information.
- Monitor system performance.
- Receive notifications about critical system events, directly or via e-mail.
- Set up Virtuozzo Virtual Networks.
- Manage Parallels Infrastructure.
- Install, update, and remove Virtuozzo templates.
- Manage operating system services.
- Manage devices.
- Manage files and directories.
- Manage users and groups.

Parallels Agent API

Parallels Agent provides two APIs that you can use to create client applications:

XML API

The XML API is a set of rules by which clients can exchange information with and request actions from Agent. The XML API protocol is based on XML messages. A message is an XML document composed of XML elements that specify the request or response parameters. Each message is defined using the XML Schema 1.0 standard.

With XML API, you compose an XML request in accordance with the schema and send it to Agent using SSL or other supported protocol. Agent processes the request, takes the appropriate action, and sends back an XML response containing the data that resulted from the request. Your application then parses the received XML to extract the data.

The XML API is described in detail in the [Using XML API](#) chapter (p. 21).

SOAP API

The Parallels Agent SOAP API is a Web service based on the SOAP 1.1 and WSDL 1.1 standards. With SOAP API, you build your client applications using one of the SOAP clients that can access a Web service. This can be a SOAP client that can generate proxy classes from the provided WSDL documents, such as Visual Studio .NET. You can also create your programs using one of the scripting languages with SOAP support, such as Perl and SOAP::Lite.

You make an API call by invoking a proxy class method in a language native format. Transparently to the programmer, the SOAP client transforms the method invocation into a SOAP message and sends it to Agent over HTTPS. Agent processes the message, takes the appropriate action, and sends a response (also a SOAP message) containing the data back to the SOAP client. The client creates an appropriate object (an instance of a class) and populates it with the data from the received SOAP message. You then extract the data from the object as you usually do in the programming language that you are using.

The SOAP API is described in detail in the [Using SOAP API](#) chapter (p. 81).

Both the SOAP API and the XML API share the same Schema, so they essentially provide the same functionality. The basic format of the input and output data is also the same in both APIs. The difference is as follows:

- The XML API provides a complete set of interfaces to perform the full range of Parallels Virtuozzo Containers management and monitoring tasks.
- The SOAP API provides a similar set of functions, with some limitations. Specifically, SOAP clients cannot invoke Agent services that require the asynchronous request processing capability. This includes the services that provide performance reports on a periodic basis, progress reports, and event notifications. You can still obtain some of that data using the on-demand functionality. For example, you can obtain the most recent performance report, or retrieve performance data from a history database.

The XML Schema on which both APIs are based is described in detail in the [Parallels Agent XML API Reference](#) guide, which is a companion to this book. You can use it as a reference when programming with either API.

System Requirements

Installation

Server side

Parallels Agent software is included in the Virtuozzo Tools package, which comes with Parallels Virtuozzo Containers software. Virtuozzo Tools are installed on your server by default during Virtuozzo Containers installation. If the Virtuozzo Tools package is not installed on your server, run the Virtuozzo Containers installation program again and choose the Virtuozzo Tools installation option.

When Agent is installed on your 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. System administrator is by default granted all access rights in Agent, which means that the user can execute any of the Agent API calls and access any of the Virtuozzo Containers on the Hardware Node. You can add more users with specific access rights later using Virtuozzo Tools or programmatically through Agent.

Client side

The only software that you'll need on your client machine is the development environment of your choice. No additional client software is required.

To run the XML API samples provided in this guide, you will need Perl installed on your development machine.

To run the SOAP API samples, you'll need Microsoft Visual Studio .NET and Microsoft .NET Framework installed.

For more information and additional system requirements, please also see the [Using XML API](#) (p. 21) and [Using SOAP API](#) (p. 81) chapters respectively.

Starting, Stopping, Restarting

Before creating and running your client applications, make sure that the Parallels Agent on your server is installed and running properly.

 On Linux, the `vzagent_ctl` command line utility is used for starting, stopping, restarting, and getting the current status of Parallels Agent. The command is executed on the server where Agent 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 Agent 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 Agent 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 Agent, the output may contain additional messages describing the problem. In such a case, try restarting the Agent service 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, Agent 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.

Location of XSD and WSDL

The Parallels Agent XSD files are not included in the Parallels Virtuozzo Containers distribution. Instead, the XML Schema is documented in the *Parallels Agent XML Reference* guide, which is a companion to this book. The guide provides specifications and descriptions of the data types, the request and response messages, and includes XML code samples. Please use it as a reference when programming with either the XML or the SOAP API.

When programming with the SOAP API, you'll need the location of the WSDL documents in order to generate proxy classes. The WSDLs can be found at the location that uses the following format, where *VERSION* is the Agent protocol version number:

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

The URL to the current version 4.0.0 is as follows:

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

Agent Architecture

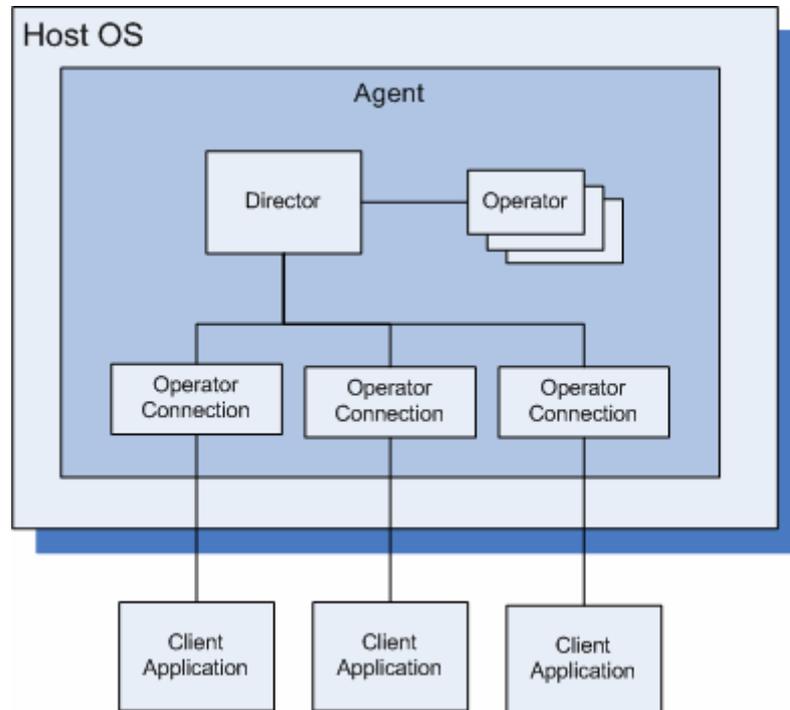


Figure 1: Agent Architecture

Parallels Agent 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 Agent architecture are *operators* and *directors*.

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

An operator is a process that is forked or spawned from a director process. There's a set of Agent operators, each of which provides a specific type of functionality. For example, the `vzaenvm` operator provides the functionality for managing Virtuozzo Containers, the `vzarelocator` operator provides the functionality for migrating a Container, etc. The diagram above illustrates the Agent component structure. A client connects to Agent through the operator *Connection* (a special operator, which is created for every client connection and which serves as 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 the message should be sent to for processing. The operator processes the message, takes the appropriate action, and generates a reply which is then routed back to the client.

The operators are divided into four major groups:

- *On-demand operators.* These operators are handling synchronous requests ("one request, one reply"). An on-demand operator 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 majority of the Agent API requests are targeted at and processed by the on-demand operators.
- *Periodic Operators (collectors).* These operators are collecting statistical data on a periodic basis and can send it to the client at the specified time intervals at the client's request. These operators use asynchronous messaging ("one request, many replies").
- *Event Reporters.* Event reporters monitor the system for critical system events, such as server configuration changes or server status changes. 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 an event takes place. The client can cancel the subscription at any time.
- *System Operator.* This operator is used to log on to Agent, manage Agent configuration, see the state of the operators and directors, verify Agent version, subscribe to an event notification service, and to perform some other system tasks.

Parallels Agent API consists of interfaces that provide access to their respective server-side operators. There's one API interface for each operator. In this guide, we will discuss some of the most commonly used operators and interfaces. For the complete list of interfaces see the Parallels Agent XML Reference guide.

Connectivity

The following table describes the connection types and protocols that your client program can use to communicate with Parallels Agent. The recommended options are indicated in the Description column.

Connection	Description
SSL over TCP/IP	This is the recommended option for permanent connections. Agent 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. Agent 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 was used in the previous versions of Virtuozzo software. It is retained for compatibility purposes only and is not officially supported.

Authentication Concepts

The first thing that a client program must do is log in to Agent using a valid user name and password. Agent uses this information to verify that the user exists in the user database (called *authentication 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. Agent uses the following authentication databases:

- *System Authentication Database.* This is the user registry of the host operating system. This basically means that you can log on to Agent using an account that exists in the operating system of the Hardware Node. In fact, when Agent 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 Agent, meaning that the user can execute any of the Agent API calls, and that the user has full access to the Hardware Node and all of its Virtuozzo Containers.
- *Parallels Internal Authentication Database.* Virtuozzo Containers software comes with its own internal authentication database. This database is used to store the Virtuozzo and Agent specific authentication information. For example, the built-in security roles used in Virtuozzo Tools are stored in this database. You can use this database to store your own Agent users. In addition, the database is used to store the Agent specific security profiles (permissions and access rights) for the users that are stored in the System Authentication Database (described above) and for the external users (described below).
- *External Authentication Database (LDAP-compliant directory).* The third authentication database type is an external LDAP-compliant directory, such as Active Directory or ADAM on Windows, or OpenLDAP on Linux. Agent can perform user authentication against an existing directory. This gives you flexibility to use existing user databases without duplicating the users in the Parallels Internal Database. The only thing that you will have to do is to create Agent security profiles for these users, which can be done through Virtuozzo Tools or programmatically through Agent. The security profiles will be stored in the Agent Internal Database and will be internally linked to the user accounts stored in the external LDAP directory. This way, you can authenticate a user against an external LDAP directory but the authorization of that user (determining the user access rights) will be performed using the user security profile in the Parallels Internal Database.

Realms

When working with the Parallels Agent API, you'll see a parameter named `realm` in the user authentication and authorization related calls. A realm represents an authentication database. It's a definition that consists of the database name, the connection parameters, and the database ID, which is called *Realm ID*. Realm definitions are stored in the Agent configuration on the Hardware Node. Before you can use an authentication database, it must be defined as a realm in the Agent configuration. At least two realm definitions are created at the time the Agent software is installed: the System Realm and the Internal Realm (p. 18). If you are planning on using an external LDAP directory as your user database, you will have to create a realm representing it first. For more info and examples, please see the [Login and Session Management](#) section (p. 46).

Authorization

Authorization in Parallels Agent is based on the concept of *security roles*. A security role is identified by its unique name and contains a list of Agent tasks that it is allowed to perform. An administrator would first create a security role granting the desired Agent 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 a property called *scope*. A scope is the logical area of a Virtuozzo system where this role assignment is allowed to operate. Scope examples include the entire Hardware Node together with Virtuozzo Containers hosted by it, a particular Virtuozzo Container, or a group of Containers.

For example, you can create a security role that can start, stop, and restart a Virtuozzo Container. 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 Containers 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 Containers specified in the scope. They will not be allowed to perform any other operations, and they will not have access to other Containers that may exist on the same host.

Terminology

This section describes some of the Agent terminology. Please take a moment to familiarize yourself with it so that you can use the documentation efficiently. The table below describes the commonly used terms:

Term	Description
Server	We use the term <i>server</i> to identify any computer, physical or virtual, in the Agent infrastructure. When we talk about a physical machine hosting Virtuozzo Containers, we may call it a Hardware Node or a host. When we talk about a Virtuozzo Container specifically, we call it a Container. Sometimes, however, we will be talking about servers in general. So, when you see the term "server", it could mean any server, physical or virtual.
Server ID	This is a globally unique ID that is assigned to any server in the Agent infrastructure. As soon as Agent is installed on a physical machine, it is assigned a Server ID. Every Virtuozzo Container that you create on it is also assigned a globally unique Server ID. The ID is guaranteed to be unique across computers and networks. Server IDs are kept internally by Agent and are used as references in all other API calls that perform operations on the servers.
Environment	Same as <i>Server</i> above. This is an obsolete term but it still can be seen in some of the Agent source code and data.
EID	Environment ID. Same as <i>Server ID</i> above. The term is obsolete but it is still being used in the Agent API code -- <code>eid</code> is the name of the parameter in calls that perform operations on physical and virtual servers.

Virtuozzo Container ID	Virtuozzo Container ID is a Virtuozzo-level ID, which is assigned to every Container when it is created. If you are familiar with the previous versions of Virtuozzo, this is the old-style "VPS ID". This ID is unique only within the context of a given Hardware Node. The ID is not to be confused with the Server ID described above, which is a universally unique Agent-level ID.
Virtuozzo group Master Node Slave Node	<p>The term <i>Virtuozzo group</i> refers to a network of servers each running its own Agent software and interconnected with each other by means of internal Agent mechanisms. The servers in such a group are organized in a hierarchical structure where there's one <i>Master Node</i> and many <i>Slave Nodes</i>.</p> <p>Master Node administers the entire group by allocating, monitoring, and controlling the group resources. Master is also capable of accessing any Slave Node in a group, meaning that once a client program is connected to the Master Node, it can send requests to any Slave Node in the group.</p>
Realm	Realm is a collection of parameters that define an authentication database containing the Agent user and group data. Agent supports a number of different databases, including operating system user registries and LDAP-compliant directories. Realm definitions are stored in the Agent configuration. Every realm is assigned a universally unique ID by Agent when it is created.

CHAPTER 3

Using XML API

The material in this chapter is intended for developers who would like to develop 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	21
Creating a Simple Client Application	35
Login and Session Management	46
Creating and Configuring Virtuozzo Containers	57
Performance Monitor	69
Events and Alerts	75
Request Routing	78

XML API Basics

This section describes the main principles of the Parallels Agent XML API and technologies it is built upon. It then provides technical details on the Parallels Agent XML message format, complete with guidelines and examples. It concludes with the description of error handling in API calls.

XML Schema

XML Schema is an XML document that defines how the XML data must be organized.

The XML Schema:

- Defines elements that can appear in a document
- Defines attributes that can appear in a document
- Defines which elements are child elements
- Defines the order of child elements
- Defines the number of child elements
- Defines whether an element is empty or can include text
- Defines data types for elements and attributes
- Defines default and fixed values for elements and attributes

The Parallels Agent XML Schema defines every 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 the Schema specifications. Parallels Agent XML API is based on the XML Schema 1.1 standard.

The Parallels Agent XML Schema files (XSDs) are not included in the Parallels Virtuozzo Containers distribution. Instead, the XML Schema is documented in the **Parallels Agent XML Reference** guide, which is a companion to this book. The guide provides specifications and descriptions of data types, request and response messages, and includes XML code samples. Please use it as a reference when programming with either the XML or the SOAP API.

Agent Messages

In order to build 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 describes how an Agent message is organized, and provides the necessary specifications and examples.

XML Message Specifications

The XML message specifications in the Agent documentation are described using tables, similar to the following example:

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

expiration	0..1	int	Custom timeout value.
}			

The information in a table is based on a corresponding XML Schema and describes the format of a request or response message, or the format of a data type.

Each row in a table represents an XML element. The elements are displayed in the order they are defined in the XML Schema.

The definitions for the table columns are as follows:

Name. Specifies an XML element name. The curly brackets represent the standard XML Schema `xs:sequence` element. 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. The following is a sample XML code, built according to this specification:

```
<login>
  <name>bXluYW11</name>
  <domain>bXlkb21haW4=</domain>
  <realm>bXlyZWFSbQ==</realm>
  <password>bXlwYXNz</password>
  <expiration>1000</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 that it must occur at least once.

A number in the second position indicates the maximum allowable number of occurrences. The `[]` (square brackets) in the second position indicate that the maximum 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.

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 the information about its usage.

XML Message Examples

The following table contains an examples of a valid Agent request message:

XML element	Description
<code><packet version="4.0.0" id="2"></code>	This is the root element of any message. The <code>id</code> attribute specifies the packet ID. The <code>version</code> attribute specifies the protocol version.
<code><target>sessionm</target></code>	The target Agent operator that the request should be sent to for processing. Note: When using the <code>system</code> operator, do not include the <code>target</code> element. The <code>system</code> operator is the only exception. All other operators require the <code>target</code> element.
<code><data></code>	The data block contains the message body.
<code><sessionm></code>	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 <code>target</code> element above).
<code><login></code>	This is the name of the API call.
<code><name>bXluYW11</name></code>	This and the following elements are the API call parameters.
<code><realm>00000000</realm></code>	Parameter.
<code><password>bXlwYXNz</password></code>	Parameter.
<code></login></code>	Closing tag.
<code></sessionm></code>	Closing tag.
<code></data></code>	Closing tag.
<code></packet></code>	Closing tag.

A response message may look similar to the following example:

XML element	Description
<code><packet id="2" time="2007-03-11T11:17:30+0000" priority="0" version="4.0.0"></code>	The root element. The <code>time</code> attribute specifies the response date and time. The <code>version</code> attribute specifies the protocol version.
<code><origin>sessionm</origin></code>	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 Agent.
<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>AQUAAAAAIBWKop..</sid>	Data.
</groups>	Closing tag.
</token>	Closing tag.
</sessionm>	Closing tag.
</data>	Closing tag.
</packet>	Closing tag.

Message Header

The two main sections of any Agent XML message is the *header* and the *body*. The header provides message routing and control information. The body of the message contains the actual request (or response) parameters and data. 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 table contains the Agent message header specification, as defined in XML Schema.

Message header specification:

Name	Min..Max	Type	Description
packet			The root element of an Agent XML message.
{			
cookie	0..1	string	User-defined information describing the message, or any other type of information. The data specified here remains unchanged during the request/response operation, i.e. if you put some data into this element in the request message, the response message will contain the same data.

target	0..[]	string	<p>In request messages, this element must contain the name of the operator to which the request should be sent for processing.</p> <hr/> <p>Note: When using the system operator, do not include the target element. The system operator is the only exception. All other operators require this element.</p> <hr/> <p>The name of the operator 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.</p> <p>Multiple targets may be specified if you are including multiple calls in a single request.</p> <p>In response messages, this element contains the name of the client that originated the request (the value is generated and used internally by Agent).</p>
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 Agent host ID. Used by Agent to determine the host address. Should be either contained in the Agent 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 request messages, use this structure to specify the server to which you want to forward the request. For example, if you are sending a request to the Agent on the Hardware Node but would like the request to be processed inside a Virtuozzo Container, specify the Server ID for the Container using the <code>dst/host</code> 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	Destination Server ID. When using the message forwarding feature, it is used for specifying the ID of the target server.
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 Agent.
}			
session		string	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 <code>sessionm/login</code> API call after a successful login.
}			

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

Attributes of the <packet> element:

Attribute	Type	Description
version	string	Parallels Agent protocol version number. The current protocol version number is 4.0.0. The older 3.0.3 protocol is also supported in Virtuozzo Containers 4.0.

id	string	<p>Packet ID. If included in a request message, the response will contain the same ID. This allows the response to be correlated with the original request. 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 side for any reason. As a rule of thumb, you should always include this element in all of your outgoing packets.</p> <p>The value should normally be a string containing an integer value, but it can also contain other characters if needed.</p>
priority	string	<p>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 value of zero is the default priority.</p> <p>Priorities range from -3000 to 3000.</p> <p>-3000 to -1000 for heavy messages.</p> <p>-999 to 999 for normal messages.</p> <p>1000 to 3000 for urgent messages.</p>
time	datetime_type	<p>The time when the packet was sent; in the ISO-8601 format: (e.g. "2007-02-04T08:55:51+0000").</p>
progress	string	<p>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.</p> <p>The possible values are:</p> <p>on -- the progress reporting is on.</p> <p>off (default if the attribute is omitted) -- the progress reporting is off.</p> <p>When you turn the progress reporting on, you must also include the id attribute (above) specifying the message ID.</p>
log	string	<p>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 Agent 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.</p> <p>The requests marked as <i>Logged Operation</i> in the XML API Reference support this feature.</p> <p>Possible values are:</p> <p>on -- the logging is turned on.</p> <p>off (default) -- the logging is off.</p>

type	int	<p>*** INTERNAL ***</p> <p>Bit field for the internal type of the message.</p> <pre>#define UNFINISHED 0x00000001 #define RESPONSE 0x00000002 #define RESCHEDULE 0x00000004 #define TIMEOUT 0x00000008</pre>
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	<p>*** INTERNAL ***</p> <p>Timeout limit for message processing. Used by an operator in determining the validity of its timeout.</p>
uid	int	<p>*** INTERNAL ***</p> <p>UID of the user sending this packet.</p>

Example:

The following is an example of an Agent 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_server_ID</host>
  </dst>
  <session>session_id</session>
</packet>
```

Message Body

Message body contains the actual request or response parameters and data. 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 Agent request message. As you already know, the `packet` element is the root element of every Agent 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 [Parallels Agent XML API 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/vz1/4.0.0/types"
xmlns:ns1="http://www.swsoft.com/webservices/vz1/4.0.0/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>vz1.40000.4.4fce28dd-0cd3-13..</session_id>
      <token xsi:type="ns2:tokenType">
        <user>AQUAAAAAIAHdKM5P0wxFE7uUMZK5QPuQAAAAA==</user>
        <groups>
          <sid>AQUAAAAAIAHdKM5P0wxFE7uUMZK5QPuQAAAAA==</sid>
          <sid>AQUAAAAAIAHdKM5P0wxFE7uUMZK5QPuQAQAAAA==</sid>
          <sid>AQUAAAAAIAHdKM5P0wxFE7uUMZK5QPuQCgAAAA==</sid>
          <sid>AQUAAAAAIAHdKM5P0wxFE7uUMZK5QPuQAqAAAA==</sid>
          <sid>AQUAAAAAIAHdKM5P0wxFE7uUMZK5QPuQAwAAAA==</sid>
          <sid>AQUAAAAAIAHdKM5P0wxFE7uUMZK5QPuQBAAAAA==</sid>
          <sid>AQUAAAAAIAHdKM5P0wxFE7uUMZK5QPuQBgAAAA==</sid>
          <sid>AQUAAAAAIAHdKM5P0wxFE7uUMZK5QPuQAAAAA==</sid>
        </groups>
      <deny_only_sids/>
    </sessionm>
  </data>
</packet>
```

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

The body of a response 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 by definition. The <OK/> means that the operation completed successfully.
- An error information, in case of a failure.

A complete XML Schema specification exists for every possible response of every Agent XML API call, and is described in the corresponding section of the [Parallels Agent XML API Reference guide](#).

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 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, one separate response for each request.

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/vz1/4.0.0/types"
xmlns:ns1="http://www.swsoft.com/webservices/vz1/4.0.0/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 Agent from a client program, it must be terminated with a binary zero character (written as '\0'). The null-terminating character is used by Agent to determine the end of the message.

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 Agent operator that generated the response. The error information consists of a numeric code and a string describing the problem. Agent has its own list of errors. The errors reported by various system utilities and the internal calls invoked by Agent operators are automatically translated to their client-level Agent 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/vz1/4.0.0/sessionm"
xmlns:ns2="http://www.swsoft.com/webservices/vz1/4.0.0/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 application that will get you started with Parallels Agent programming. We will be using Perl to write our sample program. The complete program code is included in *The Complete Program Code* section (p. 44).

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. As an example, ActivePerl for Windows is available as a free download at <http://www.activestate.com>.

A client program can communicate with Agent using the secure SLL over TCP/IP or plain TCP/IP connection. The TCP/IP module comes standard with Perl. If you would like to communicate with Agent securely, you will need the `IO::Socket::SSL` module that provides SSL support for Perl. The module can be downloaded from CPAN here: <http://search.cpan.org/~behroozi/IO-Socket-SSL-0.97/SSL.pm>.

The SSL package requires another module called `Net::SSLeay`, which can also 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.

Now that we have our development environment set up, we are ready to write our program. The program will be as basic as it can possibly be but it should suffice as an entry point into the Agent programming.

Connecting to Agent

Create a new text file named `AgentExample1.pl` and paste or type the following code into it:

```
#!/usr/bin/perl -w
#
use strict;
```

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

```
#Set $SSL_ON = 1 if you wish to use secure connection.
use constant SSL_ON => 0;

#Connection information.
#Change the IP address to your own server address.
use constant CONF_CONNECTION => {
    ip => '192.168.0.37',
    port => &SSL_ON ? 4434 : 4433,
    class => &SSL_ON ? 'IO::Socket::SSL' : 'IO::Socket::INET'
};

eval "use " . &CONF_CONNECTION->{class};
die "$@" if $@;

#Null-terminating character (packet separator).
use constant MSG_TERMINATOR => "\0";
local $/ = &MSG_TERMINATOR;
```

In the code above we create a class `CONF_CONNECTION` containing the Agent connection information. By default we are using the `IO::Socket::INET` module to communicate with Agent via plain TCP/IP. If you would like to communicate with Agent securely, set the `$SSL_ON` constant to 1. Agent is using port 4433 for plain TCP/IP connections and port 4434 for SSL connections. The `MSG_TERMINATOR` constant is a binary zero character which we'll be appending to every Agent request message (every Agent request must be null-terminated). The following code creates a socket thereby getting a connection to Agent.

```
#Create socket
print "Connecting to Agent...\n\n";
our $socket = &CONF_CONNECTION->{class}->new(
    PeerAddr => &CONF_CONNECTION->{ip},
    PeerPort=> &CONF_CONNECTION->{port},
    Proto => 'tcp',
);

unless($socket) {
    die "Connection refused: $!"
}
```

We can now read the Agent response from the socket as follows:

```
#Read the greeting message from Agent.
my $hello = $socket->getline;
chomp($hello);
print $hello;
```

Save the file now and run the program by typing `perl AgentExample1.pl` at the command prompt. You should see the output on your screen similar to the following:

```
<packet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" id="0"
priority="0" version="4.0.0">
  <origin>vzclient148-4fce28dd-0cd3-1345-bb94-3192b940fb90</origin>
  <target>agent</target>
```

```
<data>
  <ok/>
  <eid>4fce28dd-0cd3-1345-bb94-3192b940fb90</eid>
</data>
</packet>
```

If you see the above message on the screen, it means that Agent is functioning properly, and that you are connected to it. If you don't see the message, make sure that Agent is running (p. 14) and that you can ping the server from your client computer.

Let's now examine the response that we received from Agent. As you can see, it is an XML document. In fact, this is the very first message that you receive from Agent every time you connect to it. It is basically a greeting message from Agent, which means that the initial connection has been established successfully. The `eid` element contains the Server ID (p. 19) of the Hardware Node that your program is now connected to.

Logging In

Once you are connected to Agent, 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, which includes a user name, a password, and a Realm ID. 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 can be executed without being logged in. 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 the XML message to the socket that we created earlier:

```
#Write the XML message to the socket.
$socket->printf($request.&MSG_TERMINATOR);
```

Please note that we appended the binary zero character contained in the `MSG_TERMINATOR` constant to the message. Failure to do so will make the request unrecognizable to Agent. 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.
my $response = $socket->getline;
chomp($response);
print $response;
```

The response to this message will contain the complete list of Realms defined in the Agent 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>Y249dnphZ2VudCkYz1WWkw=</name>
            <realm>458d583f-f2d8-7940-a9d3-a9a3d2ec1509</realm>
          </login>
          <builtin/>
          <name>Parallels 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=vz1</base_dn>
          <default_dn>cn=users,ou=4fce28dd-0cd3-1345-bb94-
3192b940fb90,dc=vz1</default_dn>
        </realm>
        <realm>
          <builtin/>
          <name>System</name>
          <type>0</type>
          <id>00000000-0000-0000-0000-000000000000</id>
```

```

    </realm>
    <realm>
      <builtin/>
      <name>Virtuozzo Container</name>
      <type>1000</type>
      <id>00000000-0000-0000-0100-000000000000</id>
    </realm>
  </realms>
</system>
</data>
</packet>

```

Assuming that Virtuozzo Containers software has just been installed on our system, we will use the system administrator account to log in to Agent (see [Installation](#) (p. 13) for more info). The System Realm (p. 18) 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 Agent 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>bXlwYXNz</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.
$socket->printflush($request.&MSG_TERMINATOR);

```

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.
$response = $socket->getline;
chomp($response);
print $response;

```

If the supplied credentials were valid, the response message will contain the user security information, 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>AQUAAAAAIAHdKM5P0wxFE7uUMZK5QPuQAAAAA==</user>
      <groups>
        <sid>AQUAAAAAIAHdKM5P0wxFE7uUMZK5QPuQAAAAA==</sid>
        <sid>AQUAAAAAIAHdKM5P0wxFE7uUMZK5QPuQAQAAAA==</sid>
        <sid>AQUAAAAAIAHdKM5P0wxFE7uUMZK5QPuQCgAAAA==</sid>
        <sid>AQUAAAAAIAHdKM5P0wxFE7uUMZK5QPuQAQAAAA==</sid>
        <sid>AQUAAAAAIAHdKM5P0wxFE7uUMZK5QPuQAwAAAA==</sid>
        <sid>AQUAAAAAIAHdKM5P0wxFE7uUMZK5QPuQBAAAA==</sid>
      </groups>
    </system>
  </data>
</packet>

```

```
<sid>AQUAAAAAIADdKM5P0wxFE7uUMZK5QPuQBgAAAA==</sid>
<sid>AQUAAAAIAHdKM5P0wxFE7uUMZK5QPuQAAAAAA==</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 in to Agent 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 as the one we specified in the request message. The `target` element contains the ID of our client connection (the value is assigned and used by Agent internally). The `origin` element contains the name of the Agent 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 Agent will retrieve a list of the Virtuozzo Containers from the Hardware Node.

Retrieving a List of Virtuozzo Containers

To retrieve a list of Virtuozzo Containers from the Hardware Node, we will use the `get_list` call from the `vzaenvm` interface (Virtuozzo Container management).

```
#XML message. Getting a list of Virtuozzo Containers.
$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.
$socket->printflush($request.&MSG_TERMINATOR);

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

The response will contain the list of Server IDs (p. 19). 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>e9ab2834-ed97-1f4b-bd41-81c27facfc30</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, we'll add a code to our program that will restart one of the Virtuozzo Containers from the list above.

Restarting a Virtuozzo Container

The `restart` call from the `vzaenvm` interface is used to restart a Virtuozzo Container. The call accepts a single parameter: the Server ID of the Container to restart. We will use the Server ID of one of the Containers from the list that we retrieved in the previous step (p. 41).

```
#XML message. Restarting a Container.
$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.
$socket->printflush($request.&MSG_TERMINATOR);

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

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 Agent "OK" message, which is returned when an API call doesn't return any data. It simply means that the request executed successfully.

Summary

In this section, we've created a simple client demonstrating:

- 1 How to establish a connection with Agent.
- 2 How to log in to Agent.
- 3 How to retrieve a list of the Virtuozzo Containers from the Hardware Node.
- 4 How to restart a Container.

Steps 1 and 2 are the necessary steps that must be taken in any Agent application. The steps 2 and 3 have demonstrated how to work with Virtuozzo Containers. The `vzaenvm` interface is not limited to those two tasks of course. You can refer to the [Parallels Agent XML Reference](#) guide for the complete documentation of the `vzaenvm` and other interfaces.

The Complete Program Code

```
#!/usr/bin/perl -w
#
#Copyright (c) 2008 by SWsoft
#

use strict;

#Set $SSL_ON = 1 if you wish to use secure connection.
use constant SSL_ON => 0;

#Connection information.
use constant CONF_CONNECTION => {
    ip => '192.168.0.37',
    port => &SSL_ON ? 4434 : 4433,
    class => &SSL_ON ? 'IO::Socket::SSL' : 'IO::Socket::INET'
};

eval "use ".$CONF_CONNECTION->{class};
die $@ if $@;

#Null-terminating character (packet separator).
use constant MSG_TERMINATOR => "\0";
local $/ = &MSG_TERMINATOR;

#Create socket
print "Connecting to Agent...\n\n";
our $socket = &CONF_CONNECTION->{class}->new(
    PeerAddr => &CONF_CONNECTION->{ip},
    PeerPort=> &CONF_CONNECTION->{port},
    Proto => 'tcp',
);

unless($socket) {
    die "Connection refused: $!"
}

#Read the greeting message from Agent.
my $hello = $socket->getline;
chomp($hello);
print $hello;

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";
$socket->printflush($request.&MSG_TERMINATOR);

#Read the response and display it on the screen.
my $response = $socket->getline;
```

```

chomp($response);
print $response;
print "\n";
print "-----\n\n";

#XML message. Logging on.
#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>bXlwYXNz</password>
      </login>
    </system>
  </data>
</packet>
~;

#Write the XML message to the socket.
print "Logging on...\n\n";
$socket->printflush($request.&MSG_TERMINATOR);

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

#XML message. Getting a list of Virtuozzo Containers.
$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 Containers...\n\n";
$socket->printflush($request.&MSG_TERMINATOR);

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

#XML message. Restarting a Container.
#Change the Server ID to the ID of your Container.
$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 Container...\n\n";
$socket->printflush($request.&MSG_TERMINATOR);

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

Login and Session Management

The Agent login procedure comprises the following steps:

- 1 Getting a list of Realms from Agent and choosing the Realm against which to authenticate the user. You can perform this first step without being logged in to Agent.
- 2 Log in using the name and password of the user from the selected Realm. If authentication is successful, a permanent session will be created for the user.
- 3 Optionally, you may create an additional, temporary session for the user.

The following subsections describe each step in detail.

Retrieving Realm Information

To retrieve the list of the existing Realms, use the following request:

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

Once again, this call does not require you to be logged in. The Agent response will contain the list of the available Realms and will look similar to the following:

```
<packet xmlns:ns2="http://www.swsoft.com/webservices/vz1/4.0.0/types"
xmlns:ns1="http://www.swsoft.com/webservices/vz1/4.0.0/dirm"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" id="8c46e79delt18ber68c"
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>Y249dnphZ2VudCvkYz1WWkw=</name>
          <realm>458d583f-f2d8-7940-a9d3-a9a3d2ec1509</realm>
        </login>
        <builtin/>
        <name>Parallels 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=vz1</base_dn>
        <default_dn>cn=users,ou=4fce28dd-0cd3-1345-bb94-
3192b940fb90,dc=vz1</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 Container</name>
        <type>1000</type>
        <id>00000000-0000-0000-0100-000000000000</id>
      </realm>
    </realms>
  </system>
</data>
</packet>
```

The message above contains three Realm entries: Parallels Internal, System, and Virtuozzo Container. The following describes each entry in detail.

Parallels Internal Realm

```
<realm xsi:type="ns1:dir_realmType">
  <login>
    <name>Y249dnphZ2VudCvkYz1WWkw=</name>
    <realm>458d583f-f2d8-7940-a9d3-a9a3d2ec1509</realm>
```

```

</login>
<builtin/>
<name>Parallels 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=vz1</base_dn>
<default_dn>cn=users,ou=4fce28dd-0cd3-1345-bb94-
3192b940fb90,dc=vz1</default_dn>
</realm>

```

The Parallels Internal Realm is an authentication database that is installed on the host server during the Virtuozzo Containers software installation. This database is used to store the Virtuozzo Containers 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 Agent 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 not included in the Realm definition but is known to Agent. Agent uses this information to bound to the directory to perform user authentication. The empty `builtin` element indicates that this is a built-in Parallels Internal Realm (as opposed to custom Realms created by users). In fact, the rest of the Realms in this example are built-in Realms. The `address`, `port`, `base_dn`, and `default_dn` parameters describe the directory in terms of connectivity. Again, all of these elements are used by Agent to bound to the directory instance. At this point they 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 Parallels Internal Realm in your Virtuozzo Containers installation may not be the same as the ID in our example. There can be only one Parallels Internal Realm on any given Hardware Node.

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 user registry of the host operating system. When Agent is first installed, you will not have any Agent-specific users in any of the other Realms except the System Realm. If you have just started with Agent programming, use the system administrator account to log in to it. Agent knows how to identify the user with system administrator privileges and by default grants her/him unlimited access to the host server and all of the Virtuozzo Containers 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 obtain it from the Agent installed on your server. You find the System Realm record in the result set by looking at the Realm type, which should be 0 (zero).

Virtuozzo Container Realm

```

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

```

This Realm represents an operating system user registry inside a Virtuozzo Container. Use this Realm if you would like to log in to Agent as a user of one of the Containers. 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 Agent installed on your server.

External LDAP directories

In our example, we didn't have any Realms representing an external LDAP directory. These Realms are added by Virtuozzo Containers system administrators when they want to perform user authentications against an external LDAP directory. The Realm record would look similarly to the Virtuozzo Internal Realm described above except that the `builtin` parameter would not be present.

Logging In

We've demonstrated the login procedure in the beginning of this chapter when we created a sample program (p. 38). This subsection describes the procedure in detail.

The initial login is performed by executing the `system/login` request:

```
<packet version="4.0.0" id="3">
  <data>
    <system>
      <login>
        <name>cm9vdA==</name>
        <realm>00000000-0000-0000-0000-000000000000</realm>
        <password>bXlwYXNz</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, Agent will try to find the user in the host operating system user registry and will verify that the supplied credentials are correct. If we wanted to log in as a user from any other Realm, we would execute the same call supplying the appropriate user name, password, and the Realm ID.

When logging in as a user from the Virtuozzo Container Realm (another built-in Realm), the call is executed slightly differently. Let's say that we want to log in to Agent as the `root` user from one of the Containers running on the Hardware Node. This is how you do it:

```
<packet version="" id="3">
  <data>
    <system>
      <login>
        <name>cm9vdA==</name>
        <domain>ZTlhYjI4MzQtZWQ5Ny0xZjRiLWJkNDEtODFjMjdmYWNmYzZm</domain>
        <realm>00000000-0000-0000-0100-000000000000</realm>
        <password>bXlwYXNz</password>
      </login>
    </system>
  </data>
</packet>
```

Compared to the previous login example, the XML packet above contains an additional domain parameter. When logging in as a user from the Virtuozzo Container Realm, the domain element must contain the Server ID of the Container. See [Retrieving a List of Virtuozzo Containers](#) section (p. 41) for an example on how to retrieve Server IDs.

If the login is 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>AQUAAAAIAHdKM5P0wxFE7uUMZK5QPuQAAAAA==</user>
        <groups>
          <sid>AQUAAAAIADdKM5P0wxFE7uUMZK5QPuQAAAAA==</sid>
```

```
<sid>AQUAAAAAIAddKM5P0wxFE7uUMZK5QPuQAQAAAA==</sid>
<sid>AQUAAAAAIAddKM5P0wxFE7uUMZK5QPuQCgAAAA==</sid>
<sid>AQUAAAAAIAddKM5P0wxFE7uUMZK5QPuQAQAAAA==</sid>
<sid>AQUAAAAAIAddKM5P0wxFE7uUMZK5QPuQAwAAAA==</sid>
<sid>AQUAAAAAIAddKM5P0wxFE7uUMZK5QPuQBAAAA==</sid>
<sid>AQUAAAAAIAddKM5P0wxFE7uUMZK5QPuQBgAAAA==</sid>
<sid>AQUAAAAIAHdKM5P0wxFE7uUMZK5QPuQAAAA==</sid>
</groups>
<deny_only_sids/>
<privileges/>
</token>
</system>
</data>
</packet>
```

The output contains the security IDs (SIDs) of the user and all the groups to which the user belongs as a member.

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 is associated with the physical connection to Agent that your client is using. If you are not planning on logging in multiple users from the same program, you may simply use this session to execute your requests. A permanent session never expires, which means that even if your client program doesn't send any requests to Agent for a long time, the session will still stay active. When you are done working with Agent, you may simply exit and the session will be terminated automatically.

Optionally, you may create additional sessions using the `sessionm` 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 `system/login` call described in the previous section. The output will contain the SIDs plus the ID of the new 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>vz1.40000.4.4fce28dd-0cd3-1345-bb94-3192b940fb90..f7c46e51175t321d6069r202d</session_id>
      <token>
        <user>AQUAAAAIAHdKM5P0wxFE7uUMZK5QPuQAAAAA==</user>
        <groups>
          <sid>AQUAAAAAIAddKM5P0wxFE7uUMZK5QPuQAAAAA==</sid>
          <sid>AQUAAAAAIAddKM5P0wxFE7uUMZK5QPuQAQAAAA==</sid>
          <sid>AQUAAAAAIAddKM5P0wxFE7uUMZK5QPuQCgAAAA==</sid>
          <sid>AQUAAAAAIAddKM5P0wxFE7uUMZK5QPuQAQAAAA==</sid>
          <sid>AQUAAAAAIAddKM5P0wxFE7uUMZK5QPuQAwAAAA==</sid>
          <sid>AQUAAAAAIAddKM5P0wxFE7uUMZK5QPuQBAAAA==</sid>
          <sid>AQUAAAAAIAddKM5P0wxFE7uUMZK5QPuQBgAAAA==</sid>
          <sid>AQUAAAAIAHdKM5P0wxFE7uUMZK5QPuQAAAAA==</sid>
        </groups>
        <deny_only_sids/>
        <privileges/>
      </token>
    </sessionm>
  </data>
<src>
```

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

In the output above, the `session_id` element contains the new session ID. You must include this ID in every Agent request in order for the request to be processed within the contexts of the session. Failure to do so will result in the message being sent and processed using the default session created by the `system/login` call. The following example shows how to include the session ID in an Agent 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 Agent 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. For the complete list of calls provided by the `sessionm` interface, please see the [Parallels Agent XML API Reference guide](#).

Logging In To VZCC or VZPP

Using Agent API, it is possible to programmatically login to VZCC (Virtuozzo Control Center) or VZPP (Virtuozzo Power Panels). This functionality allows to access VZCC or VZPP via a Web browser control embedded in your client application.

The following is a summary describing the steps in performing VZCC/VZPP login:

- 1 Create a new Parallels Agent session.
- 2 Populate the session storage area with the appropriate data according to predefined rules.
- 3 Build a VZCC/VZPP connection URL and pass it to the Web browser.
- 4 As a result, the main VZCC/VZPP screen will open in the browser. The user will then be able to use it to manage Parallels Virtuozzo Containers in a usual manner.

The rest of this section describes how to perform each of the above steps in detail.

Creating a New Session

This step is performed using the `sessionm/login` or `sessionm/duplicate_session` call. Please don't forget that your client application must first log in to Parallels Agent (p. 50) before it can create additional sessions.

If you are logging in to VZCC, you must supply the following parameters:

- `name` - User name. The user must have permissions to access VZCC.
- `realm` - Realm ID.
- `password` - User password.

The following sample illustrates how to create a session that will be used to log in to VZCC:

```
<packet xmlns:ns1="http://www.swsoft.com/webservices/vz1/4.0.0/types"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  id="5dc458a2d5ct6de91b18r1762" version="4.0.0" type="2">
  <target>sessionm</target>
  <data>
    <sessionm>
      <login xsi:type="ns1:auth_nameType">
        <ns1:name>cm9vdA==</ns1:name>
        <ns1:realm>00000000-0000-0000-0000-000000000000</ns1:realm>
        <password>bXlwYXNz</password>
      </login>
    </sessionm>
  </data>
</packet>
```

If you are logging in to VZPP, the parameters are as follows:

- `domain` - This parameter must contain the base-64 encoded EID (Environment ID) of the desired Virtuozzo Container.
- `name` - User name. The user must be an administrator of the specified Virtuozzo Container.
- `realm` - Realm ID. This must be a Virtuozzo Container Realm (type 1000) (p. 47) referencing the operating system user registry in the specified Virtuozzo Container.
- `password` - User password.

The following sample illustrates how to create a session that will be used to log in to VZPP:

```

<packet xmlns:ns1="http://www.swsoft.com/webservices/vz1/4.0.0/types"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  id="15dac47b1c16bt6174e9d2r5fb4" version="4.0.0" type="2">
  <target>sessionm</target>
  <data>
    <sessionm>
      <login xsi:type="ns1:auth_nameType">

<ns1:domain>Yzl jNzYwNWMtOTA4ZS1iMTQ0LTkzMzgtMTBiNzAxYzFlNjdh</ns1:domain>
      <ns1:name>cm9vdA==</ns1:name>
      <ns1:realm>00000000-0000-0000-0100-000000000000</ns1:realm>
      <password>bXlwYXNz</password>
    </login>
  </sessionm>
</data>
</packet>

```

Populating The Session With Appropriate Data

This step is performed using the `sessionm/put` call. The call inserts a key/value pair into the session storage area on the server side. This data is evaluated by VZCC or VZPP during the login operation and is used to determine the user access rights. The following table describes the available keys.

Key	Type	Description
auth-type	int	Authorization type. Possible values are: 4 - Cluster user to login to VZCC. 7 - Anonymous to login to VZCC. 6 - Login to VZCC for password restore only. 11 - Login to VZPP as Container administrator. 3 - Login to VZPP with Plesk as Plesk Administrator. 7 - Login to VZPP for password restore only. 12 - Reserved SSO login.
token	tokenType	A token containing the user security information. The token is obtained from the return of the <code>sessionm/login</code> call described in the <i>Creating a New Session</i> step above. For the complete definition of the <code>tokenType</code> type, see the Parallels Agent XML Reference .
vzcp	base64Binary	Additional login information. Currently, this parameter must contain exactly the following base64-encoded XML fragment: Parallels Infrastructure Manager<login>root</login><eid>00000000-0000- 0000-0000-000000000000</eid></vzcp>

Note: All of the keys described in the table above are required and must be inserted into the session storage before attempting to log in to VZCC or VZPP. The value of the auth-type key must be selected from the list of the predefined values.

The following sample illustrates how to populate the session storage with the data:

```
<packet xmlns:ns1="http://www.swsoft.com/webservices/vz1/4.0.0/types"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  type="2" version="4.0.0">
  <target>sessionm</target>
  <data>
    <sessionm>
      <put>
        <session_id>vz1.40000.4.40ab1a76-12b8-48ce-9cda-
b82098a3334e..4c47b177ddt54cdfb74r2306</session_id>
        <data>
          <key>auth-type</key>
          <value>4</value>
        </data>
        <data>
          <key>token</key>
          <value xsi:type="ns1:tokenType">
            <ns1:user>AQUAAAAAIAF2GqtAuBLOSJzauCCYozNOAAAAAA==</ns1:user>
            <ns1:groups>
              <ns1:sid>AQUAAAAAIAB2GqtAuBLOSJzauCCYozNOAAAAAA==</ns1:sid>
              <ns1:sid>AQUAAAAAIAB2GqtAuBLOSJzauCCYozNOAQAAAA==</ns1:sid>
              <ns1:sid>AQUAAAAAIAB2GqtAuBLOSJzauCCYozNOCgAAAA==</ns1:sid>
              <ns1:sid>AQUAAAAAIAB2GqtAuBLOSJzauCCYozNOAgAAAA==</ns1:sid>
              <ns1:sid>AQUAAAAAIAB2GqtAuBLOSJzauCCYozNOAwAAAA==</ns1:sid>
              <ns1:sid>AQUAAAAAIAB2GqtAuBLOSJzauCCYozNOBAAAA==</ns1:sid>
              <ns1:sid>AQUAAAAAIAB2GqtAuBLOSJzauCCYozNOBgAAAA==</ns1:sid>
              <ns1:sid>AQUAAAAIAF2GqtAuBLOSJzauCCYozNOAAAAAA==</ns1:sid>
            </ns1:groups>
            <ns1:deny_only_sids/>
            <ns1:privileges/>
          </value>
        </data>
        <data>
          <key>vzcp</key>
          <value>PHZ6Y3A+PGxvZ2luPnJvb3Q8L2xvZ2luPjxlaWQ+MDAwMDAwMDAwMDAwMCAwMDAwLTAwMDA
tMDAwMDAwMDAwMDAwPC9laWQ+PC92emNwPg==</value>
        </data>
      </put>
    </sessionm>
  </data>
</packet>
```

Building The Connection URL

The format of the VZCC/VZPP connection URL is as follows:

```
https://IP_address/vz/cp/login-external?LoginTicket=ticket_id
```

Where `IP_address` is the IP address of the host server (VZCC) or the Virtuozzo Container (VZPP) and `ticket_id` is the ID of the session that you created in the *Creating a New Session* step above. The following is an example of a valid URL:

```
https://10.30.25.145/vz/cp/login-external?LoginTicket=vz1.40000.4.40ab1a76-
12b8-48ce-9cda-b82098a3334e..4c47b177ddt54cdfb74r2306
```

Creating and Configuring Virtuozzo Containers

This section describes how to create a Virtuozzo Container using Agent XML API.

Getting a List of Sample Configurations

When creating a Virtuozzo Container, you must choose a sample configuration for it. Virtuozzo Containers software comes with a number of sample configurations, which are automatically installed on the host server. 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 of the available configurations with the complete set of parameters for each one (the output will be very long). You can review the parameters and their values but what you really need is the configuration name and ID. The following example shows the typical output. The QoS (quality of service) and some of the other configuration 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/vz1/4.0.0/types"
xmlns:ns1="http://www.swsoft.com/webservices/vz1/4.0.0/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. You will use it later as an input parameter in the request that will create the Container.

Getting a List of OS Templates

A Virtuozzo Container is based on an Operating System template (OS template). When creating a Container, you must choose and specify the OS template name. The OS templates are shipped with Virtuozzo Containers software and are installed on the Hardware Node. To get the list of the available OS templates, use the `vzapackagem/list` call as shown in the following example:

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

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

```
<?xml version="1.0" encoding="UTF-8"?><packet
xmlns:ns3="http://www.swsoft.com/webservices/vz1/4.0.0/types"
xmlns:ns1="http://www.swsoft.com/webservices/vz1/4.0.0/package"
xmlns:ns2="http://www.swsoft.com/webservices/vza/4.0.0/vzatypes"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="4.0.0"
priority="0" id="6c486384eat2cd6r1be8" time="2008-06-26T15:49:01+0000">
<origin>vzapackagem</origin>
<target>vzclient2-67cefb4a-9a6e-2d41-94aa-cbfd20fa3943</target>
<dst>
  <director>gend</director>
</dst>
<data>
  <vzapackagem>
    <packages>
      <package xsi:type="ns2:package_vztemplateType">
        <name>.fedora-core-6-x86</name>
        <os xsi:type="ns3:osType">
          <platform>Linux</platform>
          <name></name>
        </os>
        <arch>x86</arch>
        <os_template>1</os_template>
        <cached>1</cached>
        <uptodate>0</uptodate>
      </package>
      <package xsi:type="ns2:package_vztemplateType">
        <name>.redhat-el5-x86</name>
        <os xsi:type="ns3:osType">
          <platform>Linux</platform>
          <name></name>
        </os>
        <arch>x86</arch>
        <os_template>1</os_template>
        <cached>1</cached>
        <uptodate>0</uptodate>
      </package>
      <package xsi:type="ns2:package_std_vztemplateType">
        <name>redhat-as3-minimal</name>
        <version>20061020</version>
        <os xsi:type="ns3:osType">
          <platform>Linux</platform>
```

```

    <name></name>
  </os>
  <arch>x86</arch>
  <os_template>1</os_template>
  <cached>1</cached>
  <uptodate>0</uptodate>
  <technology>npt1</technology>
  <technology>x86</technology>
  <base>1</base>
</package>
</packages>
</vzapackagem>
</data>
<src>
  <director>gend</director>
</src>
</packet>

```

Choose the OS template from the list and get its name. The template name will be used as a parameter in the call that will create the Container later. In our example, we have just one template and its name is "redhat-as3-minimal" (the standard Virtuozzo Containers Red Hat Linux template).

Populating Container Configuration Structure

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

The rest of the configuration parameters (such as disk quota, CPU parameters, etc.) can also be customized but it should only be done by the experienced users. In this example, we will set all of the parameters from the table above. We will not modify any of the advanced parameters so their values will be taken from the sample configuration file. The configuration portion of the XML request that will create our Container will look like this:

```
<config>
  <name>My-CT10</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 a Virtuozzo Container

The final step in creating a Container is to build the XML request and send it to Agent. To create a Container, use the `vzaenvm/create` call.

The following request will create a Virtuozzo Container:

```
<packet version="4.0.0" id="2">
  <target>vzaenvm</target>
  <data>
    <vzaenvm>
      <create>
        <config>
          <name>My-CT10</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 Container 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 Server ID that was assigned to the new Container by Agent, the ID of the Parent Server (the Hardware Node), and some of the Container information. If you see an output like that, it means that the Container was created successfully. You can also log in to your Hardware Node and run the `vzlist` command from the command prompt. You should see the new Container in the list.

Retrieving Container Configuration

You can retrieve the Container configuration information by executing the simple `vzaenvm/get_info` request as follows:

```
<packet version="4.0.0" id="2">
  <target>vzaenvm</target>
  <data>
    <vzaenvm>
      <get_info>
        <eid>a5961178-14d2-40cc-b1e7-41b562a2f4c6</eid>
        <config/>
      </get_info>
    </vzaenvm>
  </data>
</packet>
```

The `eid` parameter specifies the Server ID of the Container. The output will contain the complete Container information including its Parent Server ID (the ID of the Hardware Node), the status information (running, stopped, etc.), the alert information if any alerts are currently raised on the Container, and the Container configuration data. The following example is an output with most of the QoS parameters omitted for brevity:

```
<packet priority="0" version="4.0.0">
  <origin>vzaenvm</origin>
  <data>
    <vzaenvm>
      <env xsi:type="envType">
        <parent_eid>89e27960-97b8-461f-902f-557b4b16784b</parent_eid>
        <eid>3e25fee2-1163-4336-9e74-8b8097936d47</eid>
        <status xsi:type="ns3:env_statusType">
          <state>6</state>
        </status>
        <alert>0</alert>
        <config xsi:type="env_configType"/>
        <virtual_config xsi:type="venv_configType">
          <hostname>myhost</hostname>
          <name>Mycomputer</name>
          <offline_management>1</offline_management>
          <on_boot>1</on_boot>
          <os_template>
            <version>20061020</version>
            <name>redhat-as3-minimal</name>
          </os_template>
          <ve_root>/vz/root/$VEID</ve_root>
          <ve_private>/vz/private/$VEID</ve_private>
          <ve_type>
            <veid>0</veid>
            <type>1</type>
          </ve_type>
          <qos>
            <id>avnumproc</id>
            <hard>40</hard>
          </qos>
          <qos>
            <id>cpuunits</id>
            <hard>1000</hard>
          </qos>
          <qos>
            <id>dcachesize</id>
            <hard>1097728</hard>
            <soft>1048576</soft>
          </qos>
        </virtual_config>
      </env>
    </vzaenvm>
  </data>
</packet>
```

```

<!-- ..... ->

<veid>101</veid>
<type>virtuozzo</type>
<offline_service>vzpp</offline_service>
<offline_service>vzpp-plesk</offline_service>
<os xsi:type="ns3:osType">
  <platform>Linux</platform>
  <kernel>2.6.9-023stab033.6</kernel>
  <version>20061020</version>
  <name>redhat-as3-minimal</name>
</os>
<net_device xsi:type="ns4:net_vethType">
  <id>venet0</id>
  <ip_address>
    <ip>10.100.23.203</ip>
  </ip_address>
  <ns4:host_routed/>
</net_device>
<address>
  <ip>10.100.23.203</ip>
</address>
</virtual_config>
</env>
</vzaenvm>
</data>
</packet>

```

Configuring a Virtuozzo Container

To modify the configuration parameters of an existing Virtuozzo Container, use the `vzaenvm/set` request. There are two ways that a container configuration can be modified:

- By specifying the configuration parameters and their new values
- By applying the values from a sample configuration.

The following subsections describe each method in detail.

Passing parameters explicitly

The configuration parameters can be passed explicitly by specifying the parameters and the new values in the request. By using this approach, you can modify a single parameter, a set of parameters, or the entire configuration information. The request is similar to the `create` request that creates a Container. It accepts the Server ID of the Container that you would like to update, the configuration structure (`venv_configType`), and a couple of other parameters. To execute a request, first populate the configuration structure with the parameters and values that you would like to modify (all of the parameters are optional so you can include or exclude any of them) and then pass it to Agent using the `vzaenvm/set` request.

The following sample assigns a new hostname and adds a search domain to an existing Virtuozzo Container. It is also adding two DNS servers to the default `venet0` virtual network adapter.

```
<packet version="4.0.0" id="34">
  <target>vzaenvm</target>
  <data>
    <vzaenvm>
      <set>
        <eid>3288bb6b-8a49-4230-b565-6ad5521182aa</eid>
        <config>
          <hostname>myhost</hostname>
          <search_domain>ts6.com</search_domain>
          <net_device>
            <id>venet0</id>
            <nameserver>192.168.1.51</nameserver>
            <nameserver>192.168.1.52</nameserver>
          </net_device>
        </config>
      </set>
    </vzaenvm>
  </data>
</packet>
```

The following example will modify the IP address configuration for the `venet0` network adapter, which is the default virtual adapter inside a Container. This modification works in such a way that the existing IP addresses are first removed from the adapter configuration and then the passed addresses are added replacing the old ones. To add an IP address without removing the old ones, first retrieve the existing addresses, then add the new address (or addresses) to the list, and then include the entire list in the request.

```
<packet version="4.0.0">
  <target>vzaenvm</target>
  <data>
    <vzaenvm>
      <set>
        <eid>72145bf0-7562-43d4-b707-cc33d37e3f10</eid>
        <config>
          <net_device>
            <id>venet0</id>
            <ip_address>
              <ip>10.130.1.1</ip>
            </ip_address>
            <ip_address>
              <ip>10.130.1.2</ip>
            </ip_address>
            <ip_address>
              <ip>10.130.1.3</ip>
            </ip_address>
          </net_device>
        </config>
      </set>
    </vzaenvm>
  </data>
</packet>
```

```

        </ip_address>
        <host_routed/>
    </net_device>
</config>
</set>
</vzaenvm>
</data>
</packet>

```

Using values from a sample configuration

This approach allows to specify the name of the configuration parameters but their values will be taken from a sample configuration. This is useful when setting (or re-setting) the QoS-related values because a sample configuration contains the values that are fine-tuned for the type of applications that you intend to run inside the Container. Although you can modify individual parameters, it often makes sense to modify an entire parameter category. This is accomplished by specifying the category ID using the `category` element. The following table lists the categories that can be set using this approach.

Category ID	Description
general_conf	General Container parameters.
qos	Resource parameters - UBC, disk quota, CPU - all at once.
quota	Disk quota parameters.
cpu	CPU parameters.

The following sample shows how to modify an entire set of QoS parameters in a Container using the values from the specified sample configuration.

```

<packet version="4.0.0" id="654">
  <target>vzaenvm</target>
  <data>
    <vzaenvm>
      <set>
        <eid>6dbd99dc-f212-45de-a5f4-ddb78a2b5280</eid>
        <apply_config>
          <sample_conf>f8e96630-7fd8-4eee-93b2-3ad7b6b53916</sample_conf>
          <category>qos</category>
        </apply_config>
      </set>
    </vzaenvm>
  </data>
</packet>

```

Destroying a Virtuozzo Container

When you destroy a Container, all its data is removed from the Hardware Node and cannot be recovered. You can only destroy a Container that is currently stopped. To destroy a Container, execute the following request (the `eid` element contains the Server ID of the Container to be destroyed):

```
<packet version="4.0.0" id="2">
  <target>vzaenvm</target>
  <data>
    <vzaenvm>
      <destroy>
        <eid>a5961178-14d2-40cc-b1e7-41b562a2f4c6</eid>
      </destroy>
    </vzaenvm>
  </data>
</packet>
```

Performance Monitor

Performance Monitor is an operator that allows to monitor the performance of the Hardware Node and Virtuozzo Containers. By monitoring the utilization of the system resources, you can acquire an important information about your Virtuozzo system health. Performance Monitor can track a range of processes in real time and provide you with the results that can be used to identify current and potential problems. It can assist you with the tracking of the processes that need to be optimized, monitoring the results of the configuration changes, identifying the resource usage bottlenecks, and planning of upgrades.

The performance data is collected by *Periodic Collectors*, the special operators that run on the server side at all times. Periodic collectors collect the data at the predefined time intervals (several seconds) and put it into a storage buffer where it can be read by other operators. Performance Monitor is capable of obtaining this data in real time and sending it back to the client on demand or periodically. The rest of this section describes how to use Performance Monitor in your client programs.

Classes, Instances, Counters

First, we have to discuss the Performance Monitor terminology.

Performance Class

Performance class is a type of the system resource that can be monitored. This includes CPU, memory, disk, network, etc. A class is identified by ID. See **Appendix A: Performance Counters** (p. 150) for the complete list of classes. Please note that there are two separate groups of classes: one is used for monitoring Virtuozzo Containers and the other for monitoring Hardware Nodes.

Class Instance

While class identifies the type of the system resource, the term "instance" refers to a particular device when multiple devices of the same type exist in the system. For example, network in general is a class, but each network card installed in the system is an instance of that class. Each class has at least one instance, but not all classes may have multiple instances. **Appendix A: Performance Counters** (p. 150) provides information on how to obtain a list of instances for each class.

Performance Counter

Counters are used to measure various aspects of a performance, such as the CPU times, network rates, disk usage, etc. Each class has its own set of counters. Counter data is comprised of the current, minimum, maximum, and average values. For the complete list of counters see **Appendix A: Performance Counters** (p. 150).

Getting a Performance Report

Now that we know what classes, instances, and counters are, we are ready to use Performance Monitor. In this section, we will obtain a single on-demand performance report. In the section that follows, we will use the monitor to receive periodic reports.

The first step is to select the performance class for which to obtain a report. Let's say that we want to get the current network usage by a Virtuozzo Container. The name of the class is `counters_vz_net`. The names of the counters are `counter_net_incoming_bytes` and `counter_net_outgoing_bytes`. A single performance report is obtained using the `perf_mon/get` call. The XML request will look similar to the following:

```
<packet version="4.0.0" id="2">
  <target>perf_mon</target>
  <data>
    <perf_mon>
      <get>
        <eid_list>
          <eid>6d7d3a7c-b7a7-3745-b7cb-0e56205120a1</eid>
        </eid_list>
        <class>
          <name>counters_vz_net</name>
          <instance>
            <counter>counter_net_incoming_bytes</counter>
            <counter>counter_net_outgoing_bytes</counter>
          </instance>
        </class>
      </get>
    </perf_mon>
  </data>
</packet>
```

Output

The output contains the requested performance data.

```
<?xml version="1.0" encoding="UTF-8"?><packet
xmlns:ns2="http://www.swsoft.com/webservices/vz1/4.0.0/types"
xmlns:ns1="http://www.swsoft.com/webservices/vz1/4.0.0/perf_mon"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="4.0.0"
priority="0" id="5c47838ee6t3d6cre0" time="2008-01-08T14:55:36+0000">
<origin>perf_mon</origin>
<target>vzclient19-c56d51ed-cb1c-fa4c-b131-3115d3700c68</target>
<dst>
  <director>gend</director>
</dst>
<data>
  <perf_mon>
    <data xsi:type="ns2:perf_dataType">
      <eid>6d7d3a7c-b7a7-3745-b7cb-0e56205120a1</eid>
      <interval xsi:type="ns2:intervalType">
        <start_time>2008-01-08T14:55:06+0000</start_time>
        <end_time>2008-01-08T14:55:26+0000</end_time>
      </interval>
      <class>
        <name>counters_vz_net</name>
        <instance>
          <name>0</name>
          <counter>
            <name>counter_net_incoming_bytes</name>
            <value>
              <avg>0</avg>
              <min>0</min>
```

```
        <max>0</max>
        <cur>0</cur>
    </value>
</counter>
<counter>
    <name>counter_net_outgoing_bytes</name>
    <value>
        <avg>0</avg>
        <min>0</min>
        <max>0</max>
        <cur>0</cur>
    </value>
</counter>
</instance>
<instance>
    <name>1</name>
    <counter>
        <name>counter_net_incoming_bytes</name>
        <value>
            <avg>0</avg>
            <min>0</min>
            <max>0</max>
            <cur>0</cur>
        </value>
    </counter>
    <counter>
        <name>counter_net_outgoing_bytes</name>
        <value>
            <avg>0</avg>
            <min>0</min>
            <max>0</max>
            <cur>0</cur>
        </value>
    </counter>
</instance>
</class>
</data>
</perf_mon>
</data>
<src>
    <director>gend</director>
</src>
</packet>
```

Receiving Periodic Reports

In this section, we will use Performance Monitor to receive reports on a periodic basis. This functionality is provided by the `perf_mon/start_monitor` call. The following sample shows how to start the monitoring of the CPU consumption by the specified server:

```
<packet version="4.0.0" id="2">
  <target>perf_mon</target>
  <data>
    <perf_mon>
      <start_monitor>
        <eid_list>
          <eid>39f40723-b3f5-8c41-8de9-7beefd5021fe</eid>
        </eid_list>
        <class>
          <name>counters_vz_cpu</name>
          <instance>
            <counter>counter_cpu_system</counter>
          </instance>
        </class>
        <report_period>20</report_period>
      </start_monitor>
    </perf_mon>
  </data>
</packet>
```

The call above starts the monitor with the 20 second intervals, which means that the client will be receiving a report every 20 seconds. The `eid` parameter contains the Server ID of the Virtuozzo Container to monitor. We are using just one counter from the `counters_vz_cpu` class in this example. To obtain the performance data for 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 monitor multiple servers at the same time (more on that later).

The first response that we receive from Agent contains the monitor ID. We will need this ID to stop the monitor later.

```
<packet id="2" version="4.0.0">
  <origin>perf_mon</origin>
  <data>
    <perf_mon>
      <id>491d81b1-1cae-43ec-8b7c-41d873d15991</id>
    </perf_mon>
  </data>
</packet>
```

After that, the collector starts sending us the reports. The following is an example of one of the reports:

```
<packet id="2" priority="0" version="4.0.0">
  <origin>perf_mon</origin>
  <data>
    <perf_mon>
      <data>
        <eid>4d4dcb0c-9c1e-4f7c-af81-e33db3289f61</eid>
        <class>counters_cpu</class>
        <interval>
          <start_time>2006-06-20T05:01:09+0000</start_time>
          <end_time>2006-06-20T05:01:30+0000</end_time>
        </interval>
        <instance>
          <name></name>
          <counter>
```

```

    <name>counter_cpu_system</name>
    <value>
      <avg>477</avg>
      <min>477</min>
      <max>477</max>
      <cur>2504250</cur>
    </value>
  </counter>
</instance>
</data>
</perf_mon>
</data>
</packet>

```

As you can see, the report contains the current, average, minimum, and maximum values. Since this is an incremental counter (see [Appendix A: Performance Counters](#) (p. 150)), the current value represents the total system CPU time (the time spent by the processor to execute the 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.

While receiving the performance reports, you may execute other Agent calls if needed. To correlate the request and response messages, use the ID attribute. All responses that belong to a particular request will have the same message ID as the message ID of the request.

To stop the monitor, send the following message, passing the monitor ID:

```

<packet version="4.0.0" id="2">
  <target>perf_mon</target>
  <data>
    <perf_mon>
      <stop_monitor>
        <id>491d81b1-1cae-43ec-8b7c-41d873d15991</id>
      </stop_monitor>
    </perf_mon>
  </data>
</packet>

```

Monitoring Multiple Environments

Performance Monitor allows you to monitor multiple servers at the same time. For example, you can monitor a Hardware Node and its Virtuozzo Containers simultaneously. One important requirement here is that the performance classes and the counters that you will select for each server type must be compatible with all of them. This means that if you select a class from the "generic" category, it must also exist in the "virtuozzo" category. If you mix classes and counters that exist in one category but don't exist in the other, you will get unpredictable results. One way around this is to include only the names of the classes and omit the names of the counters. This way, the names of the counters will be retrieved automatically by Performance Monitor, so the reports will contain the "correct" counters for each server type, i.e the report will show different counters for different server types.

There's one more parameter that the `start_monitor` call takes: `filter`. You can use this parameter when you want to monitor all available servers of a particular type but exclude all other servers. In order to do that, do not specify Server IDs but specify the server types to exclude from monitoring. For example, if you want to monitor only the Virtuozzo Containers but you want to exclude the Hardware Node, supply the empty `eid` element and include the `filter` element containing the value "generic". If you include the empty `eid` element and don't specify a filter, then all available servers of all types will be monitored.

Events and Alerts

Event reporters are operators that monitor the system for critical system events, such as a server status change or server configuration change. They also allow to subscribe to and receive automatic notifications if an alert is raised on a server due to resource allocation problems. These operators are subscription-based, meaning that the client must subscribe to the event notification services in order to receive notifications. The following types of subscriptions are currently available:

Subscription Name	Description
<code>env_status_subscription</code>	Triggers when the status of a server changes, including state and transition changes. The event reports the status change for every server that Agent is aware of. If you subscribe for the event on the Master Node in a Virtuozzo group, you will receive the notifications about the status changes of every server in the entire group.
<code>env_config_subscription</code>	Triggers when the configuration of a server changes. If subscribed on a Master Node in a group, reports the changes across the entire group hierarchy.
<code>alerts_subscription</code>	Reports resource allocation problems such as approaching or breaking certain limits.

To subscribe to an event notification, make the following call:

```
<packet version="4.0.0" id="2">
  <data>
    <system>
      <subscribe>
        <name>subscription_name</name>
      </subscribe>
    </system>
  </data>
</packet>
```

Where *subscription_name* is the name of one of the event subscriptions from the table above. As soon as the event takes place (or an alert is raised), a message will be sent to your client program containing the event data. You recognize the event notification message by examining the value of the `target` element in the message header, which should contain the name of the subscription, i.e. the same name that you passed to the call when you subscribed to the event. Please remember that any message may have more than one `target` element; when searching for a particular target, make sure to look through all of them.

The following examples illustrates a notification message received when one of the servers was manually stopped. The message contains the ID of the server that generated the event, the text message that may be presented to the user, and the event data (old/new state and transition codes). Note that one of the `target` elements contains the same value as the one we used in the name element in the request, which is `env_status_subscription`. Please also note that the inner data structure contains the elements specific to this event type -- in this particular case, the `env_status_event` element.

Input

Subscribing to the status change events.

```
<packet version="4.0.0" id="2">
  <data>
    <system>
      <subscribe>
        <name>env_status_subscription</name>
      </subscribe>
    </system>
  </data>
</packet>
```

Output

A notification that was received after a server was shut down.

```
<packet version="4.0.0">
  <target>events_subscription</target>
  <target>env_status_subscription</target>
  <data>
    <event>
      <eid>849c9be9-5fbb-4e7d-b100-f841f86c150e</eid>
      <time>1155317636</time>
      <source></source>
      <category>env_status_subscription</category>
      <sid>XXX</sid>
      <data>
        <env_status_event>
          <eid>62ec514e-bc38-4aee-830d-cc802ee2aadd</eid>
          <new>
            <state>3</state>
          </new>
          <old>
```

```

        <state>3</state>
        <transition>5</transition>
    </old>
    </env_status_event>
</data>
<info>
    <message>
RW52aXJvbml1bnQgJWVpZCUgc3RhdHVzIGNoYW5nZWQgZnJvbSA1b2xkJSB0byAlbmV3JQ==
    </message>
    <name></name>
    <translate/>
    <parameter>
        <message>NjJlYzUxNGUtYmMzOC00YWVlLTgzMGQtY2M4MDJlZTJhYWRk</message>
        <name>eid</name>
    </parameter>
    <parameter>
        <message>Mw==</message>
        <name>new</name>
        <translate/>
    </parameter>
    <parameter>
        <message>Mw==</message>
        <name>old</name>
        <translate/>
    </parameter>
    </info>
</event>
</data>
</packet>

```

A subscription remains in effect for the duration of the session. If a client program disconnects and then re-connects again, the subscription is canceled and the client has to subscribe again. The events that might have happened during that time will be unknown to this client. However, the majority of the events are logged internally by Agent. The even log can be accessed using the `event_log` interface.

To stop receiving the event notifications, use the following call:

Input

```

<packet version="4.0.0" id="2">
    <data>
        <system>
            <unsubscribe>
                <name>subscription_name</name>
            </unsubscribe>
        </system>
    </data>
</packet>

```

Request Routing

Request routing is an Agent feature that allows to specify the target server to which a request message should be sent. There are two types of request routing that you can use in your client applications:

- *Local routing* -- allows to route a request from the Hardware Node to any of the Virtuozzo Containers.
- *Virtuozzo group routing* -- allows to route a request from the Master Node in a Virtuozzo group to any of the Slave Nodes or Virtuozzo Containers.

Local Routing

Most of the XML API calls that deal with Virtuozzo Containers have an input parameter which is used to specify the Server ID on which the operation should be performed. For example, when you start or stop a Virtuozzo Container, you pass its Server ID to the call. In contrast, calls that allow to perform operations on both the Containers and the Hardware Node are usually missing this parameter. For example, the `filer/list` call (lists files and directories) does not have the Server ID parameter. So, how do you get file listing for a particular Virtuozzo Container? That's where request routing comes in. You can tell Agent to route the request to the specified Container and execute it there instead of executing it on the Hardware Node level. You accomplish this by including the `dst/host` (destination host) parameter in the Agent request message header to contain the Server ID of the target Container. By not including the `dst/host` parameter in the message header, you are instructing Agent to perform the operation on the Hardware Node itself. The following samples illustrate how to use the request routing feature.

In the first sample, the request message does not have the request routing information, so the response packet will contain a list of files located in the specified folder on the Hardware Node.

Input

```
<packet version="4.0.0">
  <target>filer</target>
  <data>
    <filer>
      <list>
        <cwd>Lw==</cwd>
        <path>Lw==</path>
      </list>
    </filer>
  </data>
</packet>
```

The request message in the second sample has the request routing information. The destination Server ID is included in the request using the `dst/host` element in the message header. As a result, the request will be sent to the specified Container. The result will then be routed back to the client and will contain a list of files located in the specified folder of the specified Virtuozzo Container.

```
<packet version="4.0.0">
  <dst>
    <host>3b8f950a-981d-b94d-bde1-647df39674f1</host>
  </dst>
  <target>filer</target>
  <data>
```

```

<filer>
  <list>
    <cwd>Lw==</cwd>
    <path>Lw==</path>
  </list>
</filer>
</data>
</packet>

```

When exactly do you use request routing? Here are a few simple rules:

- Use request routing if you want to perform an operation on a Virtuozzo Container but the API call that you want to use doesn't have an input parameter to specify the Server ID.
- Don't use request routing if a call has a parameter to specify the Server ID. If you try to route such a request to a Container by mistake, it will fail with a message saying that this functionality is not supported.

There are only a few interfaces in the Agent XML API that utilize the request routing functionality. Here's the list:

Class name	Description
computerm	Computer management. Provides methods for managing Hardware Nodes and Virtuozzo Containers as if they were regular physical machines.
filer	Provides methods for managing files and directories on Hardware Nodes and Virtuozzo Containers.
firewallm	Firewall management (Linux only).
processm	System processes management. Provides methods for managing system processes and for executing programs on Hardware Nodes and Virtuozzo Containers.
servicem	Services management. Provides calls for managing the operating system services on Hardware Nodes or Virtuozzo Containers.
userm	Provides calls for managing users and groups on Hardware Nodes and Virtuozzo Containers.

To use request routing in your client applications, you don't have to manually install Agent inside a Container. All the necessary Agent components are installed in a Container automatically when it is created.

Request routing in a Virtuozzo group

In a Virtuozzo group the request routing feature can be used to specify the target Slave Node or any of the Virtuozzo Containers. For example, if you are connected to the Master Node in a Virtuozzo group but would like to get the list of Virtuozzo Containers from a particular Slave Node, you can do that by routing the request to that Server using the `dst/host` parameter.

If the destination server is a Slave Node (physical machine), you can route any request to it. In this case the list of the interfaces that can utilize the request routing functionality is not limited to the short list that we've included earlier in this section. If the destination is a Virtuozzo Container, use request routing only with the interfaces listed in the table above.

Please note that you can only route Agent requests to other Nodes in a Virtuozzo group if you are connected to the Master Node. You cannot route requests between Slave Nodes. Please also note that request routing in a Virtuozzo group can be an expensive operation. The destination server can be located deep inside the group hierarchy, so by the time the message travels to its destination and back, a significant amount of the group resources may be used. Don't overuse it. Whenever possible, instead of using request routing in a Virtuozzo group, try connecting to a Slave Node directly and execute the call there.

CHAPTER 4

Using SOAP API

The material in this chapter is intended for developers who would like to develop client applications using SOAP API. To use this documentation productively, you should have a basic idea of what SOAP is, some 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.....	81
Creating a Simple Client Application	82
Developing Agent SOAP Clients.....	96
Managing Virtuozzo Containers	101
Other SOAP Clients and Their Known Issues	142
Troubleshooting	144

Introduction

This section provides an introduction to the Agent SOAP API.

Overview

Parallels Agent SOAP API is implemented as industry-standard Web Services. With SOAP API, you build your client applications using one of the third-party development tools that can generate client code from the provided WSDL documents. The code generated from WSDLs 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 XML Schema with the Agent XML API, so the basic structure of the input and output data is the same in both APIs. The [Using XML API](#) chapter (p. 21) provides general information on the Agent XML schema, the detailed description of the XML API request and response packets, and other important information. The [Parallels Agent XML Programmer's Reference](#) guide provides a complete XML API reference. When working with SOAP API, use the XML API reference material to find the descriptions of the calls, their input and output parameters, and XML code examples.

Key Features

The following describes the key SOAP API features:

- Supports the full set of the Agent on-demand functionality.
- Provides WSDL documents for automatic code generation.
- Supports a variety of third-party SOAP clients, including Microsoft .NET Framework.
- Supports SOAP 1.1 and WSDL 1.1.

Limitations

SOAP API in the Agent protocol version 4.0.0 has the following limitations:

- Operates only over the HTTPS protocol.
- Does not support asynchronous request processing.

Generating Client Code from WSDL

When programming with the SOAP API, you will need the location of the WSDL documents in order to generate proxy classes. The WSDLs can be found at the location that uses the following format, where *VERSION* is the Agent protocol version number:

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

The URL to the current version 4.0.0 is as follows:

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

Your SOAP client should have a documentation describing how to generate proxy classes from WSDL. Please follow the instructions and supply the URL of the Agent WSDL documents when asked to do so. If you are using Microsoft Visual Studio .NET, then you will find instructions on how to generate and use the code in the **Creating a Simple Client** section of this guide (p. 82).

Creating a Simple Client Application

This section walks you through the basics of creating a simple client application using Agent SOAP API. 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 sample program, let's select C# Windows console application project and call it `VzSimpleClient`.

- 1 In Microsoft Visual Studio .NET, select **File > New > Project**. The **New Project** window opens.
- 2 In the **Project Types** tree, select **Visual C# > Windows** and then select **Console Application** in the **Templates** pane.
- 3 Enter `VzSimpleClient` as the name for your project and choose a location for your project files and click **OK**.

Note: If you are using Microsoft Visual Studio .NET 2005 and if your default project files location is set to `C:\Documents and Settings\user_name\My Documents\Visual Studio 2005\Projects\project_name\..`, you will have to choose a location with a shorter path. The reason is that there's an issue with Visual Studio 2005 C# method generation from WSDL (we will discuss the issue in detail in the **Generating Stubs From WSDL** section). As a solution, we will create a batch file that will fix the problem. The file will be placed into and run from the directory that contains the **Web References** folder (usually `..\Projects\project_name\project_name\`), but because of the 256 character command line limit imposed by the Microsoft NTFS file system, the full pathname (including the path and the file name) must fit within this limit or the C# compiler will not be able to run the batch file.

Step 2: Generating Proxy Classes From WSDL

- 1 In the **Solution Explorer** pane, select the `VzSimpleClient` project.
- 2 On the **Project** menu, select **Add Web Reference**. The **Add Web Reference** window opens.

In the **URL** field, type (or copy and paste) this URL:
`http://www.swsoft.com/webservices/vza/4.0.0/VZA.wsdl`

- 1 Press the **Go** button next to the **URL** field. Visual Studio will try to connect to the SWsoft web site and retrieve the Agent web service information. After a few seconds (depends on the connection speed), you should see a single entry in the **Web services found at this URL** list box: `1 Service Found: - VZA`
- 2 Type `VZA` in the **Web reference name** field replacing the default value (in general, you can choose any name that you like). This name will be used in your code as the C# namespace to access the selected service.

Press the **Add Reference** button. This will generate proxy classes from the WSDL specifications and will add them to the project. A new item `VZA` will appear in the **Solution Explorer** in the **Web References** folder. You can now start using generated classes to access Agent services.

Step 3: Main Program File

At this point, you should see the `Program.cs` file opened in your Visual Studio IDE. This is the main file where we will write our program code. The file should contain the following code:

```
using System;
using System.IO;
using System.Collections.Generic;
using System.Text;
using VzSimpleClient.VZA;

namespace VzSimpleClient
{
    class Program
    {
        static void Main(string[] args)
        {
            // Wait for the user to press a key, then exit.
            Console.Read()
        }
    }
}
```

We've added the necessary `using` directives and we've also added the `Console.Read()` line to the `Main()` function to keep the console window open until a keyboard key is pressed.

Certificates Policy Preparation

Since Agent 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.

```
///<summary>
/// Sample class TrustAllCertificatePolicy.
/// Used as a certificate policy provider.
/// Allows all certificates.
///</summary>
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)
    {
        return true;
    }
}
```

Connection URL

An Agent server listens for secure HTTPS requests on port 4646. The connection URL will look similar to the following example (substitute the IP address value with the address of your server):

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

You may also communicate with Agent using HTTP. In this case, the port number is 8080 and the URL should look like this:

```
http://192.168.0.218:8080
```

The URL will be used as an input parameter during the login procedure described in the following step.

SOAP Object Binding

In order to send SOAP messages, we will need a helper class that will initialize type binding. In particular, this class will provide methods allowing to set up an Agent message header containing the URL, the session ID, and the target operator name.

```
public class Binder
{
    // Method to bind types.
    // Parameters
    // bindingType: Object name.
    // target: If set to true, will add the "target"
    // argument to the message header.
    // If set to false, will omit the "target"
    // argument.
    // "Target" is the name of the Agent
    // operator that processes a particular
    // request type on the server side.
    // For some requests, this argument
    // must be omitted.
    //
    public System.Object InitBinding(System.Type bindingType, bool target)
    {
        string typeName = bindingType.Name;
        System.Object Binding =
        bindingType.GetConstructor(System.Type.EmptyTypes).Invoke(null);
        bindingType.GetProperty("Url").SetValue(Binding, URL, null);
        packet_headerType header = new packet_headerType();
        header.session = session;
        if (target)
        {
            header.target = new string[1];
            header.target[0] = typeName.Replace("Binding", "");
        }
        bindingType.GetField("packet_header").SetValue(Binding, header);
        return Binding;
    }

    // Same as above, but will add the "target" argument to the
    // message header by default.
    public System.Object InitBinding(System.Type bindingType)
    {
        string typeName = bindingType.Name;
        System.Object Binding =
        bindingType.GetConstructor(System.Type.EmptyTypes).Invoke(null);
        bindingType.GetProperty("Url").SetValue(Binding, URL, null);
        packet_headerType header = new packet_headerType();
        header.session = session;
        header.target = new string[1];
        header.target[0] = typeName.Replace("Binding", "");
        bindingType.GetField("packet_header").SetValue(Binding, header);
        return Binding;
    }

    public Binder(string url, string sess)
    {
        URL = url;
        session = sess;
    }

    string URL;
    string session;
}
```

Logging in and Creating a Session

The following is an example of a function that logs the user in using the supplied connection and login parameters.

Sample function parameters:

Name	Description
<code>url</code>	Agent server URL. See the Connection URL section.
<code>name</code>	User name. We will be login in as a system administrator of the host server (Hardware Node). You will need to know the password of your Hardware Node administrator account.
<code>domain</code>	We are not going to use this parameter in this example. For more information on its usage, see Parallels Agent XML Programmer's Reference Guide .
<code>realm</code>	Realm ID. Realm is a database containing user authentication information. Agent supports various types of authentication databases, including operating system user registries and LDAP-compliant directories, such as AD/ADAM on Windows and OpenLDAP on Linux. In our example, we will be using the user registry of the Hardware Node, which is called <i>System Realm</i> in Agent terminology. The globally unique ID that Agent uses for the System Realm is 00000000-0000-0000-0000-000000000000.

The function authenticates the specified user and, if the supplied credentials are valid, creates a session for the user and returns the session ID. All subsequent Agent requests must include the session ID in order to be recognized by Agent.

Sample function:

```

/// <summary>
/// Sample function Login.
/// Authenticates the user using the specified credentials and
/// creates a new session.
/// </summary>
/// <param name="url">Agent server URL.</param>
/// <param name="name">User name.</param>
/// <param name="domain">Domain.</param>
/// <param name="realm">Realm ID.</param>
/// <param name="password">Password</param>
/// <returns>New session ID.</returns>
///
public string Login(string url, string name, string domain, string realm,
string password)
{
    try {
        System.Net.ServicePointManager.CertificatePolicy = new
TrustAllCertificatePolicy();

        // Login information object.
        login1 loginInfo = new login1();

        /* The sessionmBinding class provides the login and
        * session management functionality.
        */
        sessionmBinding sessionm = new VZA.sessionmBinding();

        /* Instantiate the System.Text.Encoding class that will
        * be used to convert strings to byte arrays.
        */
        System.Text.Encoding ascii = System.Text.Encoding.ASCII;

        // Populate the connection and the login parameters.
        sessionm.Url = url;
        loginInfo.name = ascii.GetBytes(name);
        if (domain.Length != 0) {
            loginInfo.domain = ascii.GetBytes(domain);
        }
        if (realm.Length != 0) {
            loginInfo.realm = realm;
        }
        loginInfo.password = ascii.GetBytes(password);

        // Log the specified user in.
        return sessionm.login(loginInfo).session_id;
    }
    catch (Exception e) {
        return "Exception: " + e.Message;
    }
}

```

Retrieving a List of Virtuozzo Containers

The following function retrieves a list of Virtuozzo Containers from the Hardware Node. The function accepts a numeric code specifying the Container state as a parameter allowing you to retrieve the information only for the Containers in a particular state (running, stopped, etc.). The state codes are as follows:

Code	Name
0	Unknown
1	Unexisting
2	Config
3	Down
4	Mounted
5	Suspended
6	Running
7	Repairing
8	License Violation

The function returns a string containing the list of names of the existing Virtuozzo Containers.

```

/// <summary>
/// sample function GetVEList.
/// Retrieves the list of Virtuozzo Containers from the Hardware Node.
/// </summary>
/// <param name="state">Container state code.</param>
/// <returns>Container names.</returns>
///
public string GetVEList(int state)
{
    string list_result = "";

    try {
        // Instantiate the proxy class.
        vzaenvmBinding env =
(vzaenvmBinding)binder.InitBinding(typeof(vzaenvmBinding));

        // The main input object.
        get_list1 velist = new get_list1();

        /* Set the Container status parameter.
        * -1 means ignore the status.
        */
        env_statusType[] env_status = new env_statusType[1];
        env_status[0] = new env_statusType();
        if (state == -1) {
            env_status[0].stateSpecified = false;
        }
        else {
            env_status[0].state = state;
        }
        velist.status = env_status;

        /* Get the list of the Containers then loop through it getting the
        * Server ID and the name for each Container
        */
        foreach (string ve_eid in env.get_list(velist)) {
            get_info2 ve_info = new get_info2();
            ve_info.eid = new string[1];
            ve_info.eid[0] = ve_eid;

            /* Get the Container name from the Container configuration
            structure.
            * Please note that if the name was not assigned to a
            * Container when it was created, the "name" field will be empty.
            */
            list_result += env.get_info(ve_info)[0].virtual_config.name +
"\n";
        }
    }
    catch (Exception e) {
        list_result += "Exception: " + e.Message;
    }
    return list_result;
}

```

Step 4: Running the Sample

You can build and run the program now. From the main menu, select **Build** and then **Build Solution**. Then select **Debug -> Start** (or **Start without Debugging**) to run the sample.

Complete Program Code

```

using System;
using System.IO;
using System.Collections.Generic;
using System.Text;
using VzSimpleClient.VZA;

namespace VzSimpleClient
{
    class Program
    {
        Binder binder; // Binder object variable.
        string session_id = ""; // Agent session ID.

        // Main.
        static void Main(string[] args)
        {
            Program vzClient = new Program();
            try {
                vzClient.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 xmllex) {
                Console.WriteLine(xmllex.ToString());
            }
            catch (System.InvalidOperationException opex) {
                Console.WriteLine(opex.Message + "\n" + opex.InnerException);
            }
            Console.WriteLine("Press Enter to conintinue...");
            Console.Read();
        }

        ///<summary>
        /// Sample class TrustAllCertificatePolicy.
        /// Used as a certificate policy provider.
        /// Allows all certificates.
        ///</summary>
        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)
            {
                return true;
            }
        }

        /// <summary>
        /// Sample class Binder.
        /// Provides methods to create the specified binding object
        /// and to populate the Agent message header.
        /// </summary>
        public class Binder
        {
            string URL; // Agent server URL.
            string session; // Agent session ID.

```

```

// Constructor. Sets URL and session ID values.
public Binder(string url, string sess)
{
    URL = url;
    session = sess;
}

/// <summary>
/// Method InitBinding (overloaded).
/// Creates a binding object.
/// <param name="bindingType">
/// The name of the proxy class from which to
/// create the object.
/// </param>
/// <returns>
/// <para>New binding object.</para>
/// </returns>
/// </summary>
public System.Object InitBinding(System.Type bindingType)
{
    System.Object Binding =
bindingType.GetConstructor(System.Type.EmptyTypes).Invoke(null);

    // Set URL.
    bindingType.GetProperty("Url").SetValue(Binding, URL, null);

    // Create the request message header object.
    packet_headerType header = new packet_headerType();

    // Set session ID.
    header.session = session;

    /* Set the "target" parameter in the Agent request
    * message header. The parameter must contain the name
    * of the corresponding Agent operator.
    * The operator name can be obtained from the name of the
    * proxy class. It is the substring from the beginning of the
name
    * followed by the "Binding" substring. For example, the name
is
    * of the corresponding operator for the "filerBinding" class
    * "filer".
    * All Agent requests except "system" requests must have the
requires
    * target operator value set. System is the only operator that
    * the omission of the "target" parameter from the header.
    */
    if (bindingType != typeof(systemBinding)) {
        header.target = new string[1];
        header.target[0] = bindingType.Name.Replace("Binding",
name
    "");
    }

    // Set the request message header.
header);
    bindingType.GetField("packet_header").SetValue(Binding,
header);
    return Binding;
}

/// <summary>
/// Method InitBinding (overloaded).
/// Creates a binding object.
/// Allows to set destination Container.
/// </summary>
/// <param name="bindingType">
/// The name of the proxy class from which

```

```

        /// to create the object.
        /// </param>
        /// <param name="eid">
        /// The Server ID of the destination Container to which to route
        /// the request message for processing.
        /// </param>
        /// <returns>
        /// <para>New binding object.</para>
        /// </returns>
        /// </returns>
        public System.Object InitBinding(System.Type bindingType, string
eid)
        {
            System.Object Binding =
bindingType.GetConstructor(System.Type.EmptyTypes).Invoke(null);

            // Set URL.
            bindingType.GetProperty("Url").SetValue(Binding, URL, null);

            // Create the request message header object.
            packet_headerType header = new packet_headerType();

            // Set session ID.
            header.session = session;

            /* Set the "target" parameter in the Agent request
            * message header.
            */
            if (bindingType != typeof(systemBinding)) {
                header.target = new string[1];
                header.target[0] = bindingType.Name.Replace("Binding",
"");
            }

            // Set the destination server ID.
            header.dst.host = eid;

            // Set the request message header.
            bindingType.GetField("packet_header").SetValue(Binding,
header);
            return Binding;
        }
    }

    /// <summary>
    /// Sample function Login.
    /// Authenticates the user using the specified credentials and
    /// creates a new session.
    /// </summary>
    /// <param name="url">Agent server URL.</param>
    /// <param name="name">User name.</param>
    /// <param name="domain">Domain.</param>
    /// <param name="realm">Realm ID.</param>
    /// <param name="password">Password</param>
    /// <returns>New session ID.</returns>
    ///
    public string Login(string url, string name, string domain, string
realm, string password)
    {
        try {
            System.Net.ServicePointManager.CertificatePolicy = new
TrustAllCertificatePolicy();

            // Login information object.
            login1 loginInfo = new login1();

```

```

    /* The sessionmBinding class provides the login and
    * session management functionality.
    */
    sessionmBinding sessionm = new VZA.sessionmBinding();

    /* Instantiate the System.Text.Encoding class that will
    * be used to convert strings to byte arrays.
    */
    System.Text.Encoding ascii = System.Text.Encoding.ASCII;

    // Populate the connection and the login parameters.
    sessionm.Url = url;
    loginInfo.name = ascii.GetBytes(name);
    if (domain.Length != 0) {
        loginInfo.domain = ascii.GetBytes(domain);
    }
    if (realm.Length != 0) {
        loginInfo.realm = realm;
    }
    loginInfo.password = ascii.GetBytes(password);

    // Log the specified user in.
    return sessionm.login(loginInfo).session_id;
}
catch (Exception e) {
    return "Exception: " + e.Message;
}
}

/// <summary>
/// sample function GetVEList.
/// Retrieves the list of Virtuozzo Containers from the Hardware Node.
/// </summary>
/// <param name="state">Container state code.</param>
/// <returns>Container names.</returns>
///
public string GetVEList(int state)
{
    string list_result = "";

    try {
        // Instantiate the proxy class.
        vzaenvmBinding env =
(vzaenvmBinding)binder.InitBinding(typeof(vzaenvmBinding));

        // The main input object.
        get_list1 velist = new get_list1();

        /* Set the Container status parameter.
        * -1 means ignore the status.
        */
        env_statusType[] env_status = new env_statusType[1];
        env_status[0] = new env_statusType();
        if (state == -1) {
            env_status[0].stateSpecified = false;
        }
        else {
            env_status[0].state = state;
        }
        velist.status = env_status;

        /* Get the list of the Containers, then loop through it
        * Server ID and the name for each Container
        */
        foreach (string ve_eid in env.get_list(velist)) {
            get_info2 ve_info = new get_info2();

```

```
        ve_info.eid = new string[1];
        ve_info.eid[0] = ve_eid;

        /* Get the Container name from the Container configuration
structure.
        * Please note that if name was not assigned to a
        * Container when it was created, the "name" field will be
empty.
        */
        list_result +=
env.get_info(ve_info)[0].virtual_config.name + "\n";
    }
    }
    catch (Exception e) {
        list_result += "Exception: " + e.Message;
    }
    return list_result;
}

/// <summary>
/// The Run() function is called from Main().
/// It contains the code that executes other sample functions.
/// </summary>
///
public void Run()
{
    /* The Agent server URL. Use the IP of
    * your own Hardware Node here.
    */
    string url = "http://10.30.67.54:8080/";

    // User name.
    string user = "root";

    // Domain name.
    string domain = "";

    /* Realm ID.
    * We are using the "system" realm here, so the
    * user will be authenticated against the
    * host operating system user registry.
    */
    string realm = "00000000-0000-0000-0000-000000000000";
    string password = "1q2w3e";

    // Log the user in.
    session_id = this.Login(url, user, domain, realm, password);
    Console.WriteLine("Session ID: " + session_id);
    Console.WriteLine();

    // Create the Binder object.
    if (binder == null) {
        binder = new Binder(url, session_id);
    }

    // Get the list of Containers from the Hardware Node.
    Console.WriteLine(GetVEList(-1));
    Console.WriteLine();
}
}
}
```

Developing Agent SOAP Clients

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 API Reference

SOAP API shares XML Schema with the XML API. The WSDL documents from which you generate your client code are based on the XML Schema and contain the same interfaces and calls. The .NET SOAP client generates the key classes by adding the `Binding` postfix to the original interface name. For example, the `envm` interface becomes the `envmBinding` class in C#, the `vzaenvm` interface becomes `vzaenvmBinding`, and so forth. Each class will have methods for performing specific tasks. These methods are the C# equivalents of the XML API calls from the corresponding XML API interfaces. For example, if you compare the `envmBinding` class methods with the `envm` XML API interface calls, you will see that the two sets match. What this means is that the information provided in the **Parallels Agent XML Reference** guide, describing interfaces and calls, equally applies to generated C# classes and methods. You can use this information as a reference when developing your SOAP applications.

Optional Elements

Many parameters that you supply to Agent API calls or receive from Agent are defined in the XML schema as optional elements. This means that when composing a request message, you include an element or omit it depending on the operation that you are trying to perform. In response messages, an optional element may be similarly included or not. Unfortunately, unlike the XML Schema optional elements, the class members in traditional programming languages cannot be "optional" and therefore are handled differently in this respect. The proxy classes generated from WSDL will have optional elements as primitive types (`int`, `bool`, etc.), complex types (strings, classes, structures), and arrays. The following describes how to handle each element type in your code.

Primitive Types

A primitive type member is usually flagged by a corresponding member of type `bool` declared just below it. The name of the boolean variable is made of the name of the principal member 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 its value was set when the response was generated on the server side. 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 set and it contains a meaningful value. If the value is `false` then `uid` was not set and therefore 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 `xxxSpecified` flag to `true` then the receiving code will evaluate the corresponding optional element as "not included in the request" and will ignore its value.

Complex Types

The XML schema 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 the Agent 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 like this `<some_element />`, or you simply omit it.

In C#, when passing an object of this kind to a method, you have to create it as an empty object like this:

```
some_element myObject = 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 Agent protocol are base64-encoded. 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);
```

Timeouts

Microsoft .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 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 be 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 newly generated stubs. At this point you can remove or comment out the entry that we used in the **Project > Properties > Pre-build Event Command Line** option. If that's not done, the stubs will be re-generated every time you build your solution.

If you decide to update the client code from WSDLs, repeat the described steps.

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

Managing Virtuozzo Containers

The material in this section provides code examples that demonstrate how to perform the most common Virtuozzo Containers management tasks.

Creating a Container

When creating a new Virtuozzo Container, the following configuration parameters are mandatory and must be selected every time:

- **Sample configuration name.** Virtuozzo Containers software comes with a set of sample configurations that are installed on the Hardware Node at the time the software is installed. XML API provides the `env_sample/get_sample_conf` call to retrieve the list of the available configurations. In the example provided in this section, the C# equivalent of that call is the `env_sampleBinding.get_sample_conf()` method.
- **Virtuozzo OS Template.** The list of the available templates can be retrieved using the `vzapackage.get_list` XML API call. The C# equivalent is `vzapackageBinding.get_list` call. For simplicity, we are not including this call in the example because Virtuozzo for Windows currently comes with just one OS template, and Virtuozzo for Linux has one template for each supported Linux distribution. For example, the standard Red Hat Linux OS template name is `redhat-as3-minimal`.

The rest of the parameters that we use in this example are optional but are typically used when a new Container is created. The following sample shows how to create a Virtuozzo Container

Sample Function Parameters:

Name	Description
<code>name</code>	The name that you would like to use for the Container.
<code>os_template</code>	The name of the OS template from which to create a Container.
<code>platform</code>	Operating system type: <code>linux</code> or <code>windows</code> . This parameter will be used in our function to select a sample configuration for the Container. If the sample configuration is compatible with the specified platform, we will use it. In a real application, you would probably select the sample configuration in advance and would pass its name to the method that actually creates a Container. In this example, we automate this task while providing a demonstration of how to retrieve the list of the available sample configurations.
<code>architecture</code>	CPU architecture, e.g. <code>x86</code> , <code>ia64</code> . This parameter, together with the <code>platform</code> parameter (above) will also be used to determine the sample configuration compatibility with the specified CPU architecture.
<code>hostname</code>	The hostname that you would like to use for the Container.
<code>ip</code>	The IP address to assign to the Container.
<code>netmask</code>	Container network netmask.
<code>network</code>	Network interface ID: <code>venet0</code> for Linux; <code>venet1</code> for Windows. These are the standard host-routed Virtuozzo network interfaces. For other network configuration scenarios, please refer to the Parallels Agent XML Programmer's Reference guide .
<code>offline_management</code>	Specifies whether to turn the Offline Management feature on or off.

Sample Function:

```

/// <summary>
/// Sample function CreateVE.
/// Creates a new Virtuozzo Container.
/// </summary>
/// <param name="name">Container name.</param>
/// <param name="os_template">OS template name.</param>
/// <param name="platform">Operating system type: linux or windows.</param>
/// <param name="architecture">CPU architecture (x86, ia64)</param>
/// <param name="hostname">Container hostname.</param>
/// <param name="ip">Container IP address.</param>
/// <param name="netmask">Netmask.</param>
/// <param name="network">Network interface ID.</param>
/// <param name="offline_management">
/// A flag specifyin whether to turn the "offline management"
/// feature on or off.
/// </param>
/// <returns>Server ID of the new Container.</returns>
public string CreateVE(string name, string os_template, string platform,
string architecture, string hostname, string ip, string netmask, string
network, bool offline_management)
{
    try {
        // Instantiate the proxy class.
        vzaenvmBinding env =
(vzaenvmBinding)binder.InitBinding(typeof(vzaenvmBinding));

        // The main input object.
        create create_input = new create();

        // Container configuration information.
        venv_configType1 veconfig = new venv_configType1();

        /* Retrieve the list of sample configurations.
        * Select the first one that is compatible with the
        * specified platform (Linux, Windows) and CPU architecture.
        */
        env_samplemBinding env_sample =
(env_samplemBinding)binder.InitBinding(typeof(env_samplemBinding));
        get_sample_conf get_sample = new get_sample_conf();
        sample_confType[] samples = env_sample.get_sample_conf(get_sample);

        if (samples != null) {
            foreach (sample_confType sample in samples) {
                if (sample.env_config.os != null) {
                    if (sample.env_config.os.platform == platform &&
sample.env_config.architecture == architecture) {
                        //Set Container sample
                        veconfig.base_sample_id = sample.id;
                        break;
                    }
                }
            }
        }

        // Set OS template.
        templateType osTemplate = new templateType();
        osTemplate.name = os_template;
        veconfig.os_template = osTemplate;

        // Set Container name
        veconfig.name = name;

        // Set Container hostname
        veconfig.hostname = hostname;
    }
}

```

```
// Set Container IP address and netmask.
ip_addressType[] ip_address = new ip_addressType[1];
ip_address[0] = new ip_addressType();
ip_address[0].ip = ip;
ip_address[0].netmask = netmask;

// Set network.
net_vethType[] net = new net_vethType[1];
net[0] = new net_vethType();
net[0].host_routed = new object();
net[0].id = network;
net[0].ip_address = ip_address;
veconfig.net_device = net;

// Set the offline management feature.
veconfig.offline_managementSpecified = true;
veconfig.offline_management = offline_management;

// Finalize the new Container configuration.
create_input.config = veconfig;

// Create the Container.
return env.create(create_input).env.eid;
}
catch (Exception e) {
    return "Exception: " + e.Message;
}
}
```

The function invocation example:

```
createVE( "sample_ve", "redhat-as3-minimal", "linux","x86",
"sample_ve_hostname", "10.16.3.179", "255.255.255.0", "venet0", true );
```

Starting, Stopping, Restarting a Container

To start a Container, use the `vzaenvmBinding.start()` method passing the Container ID. See [Creating a Simple Client Program](#) for an example on how to obtain the list of the IDs from the Hardware Node.

```

/// <summary>
/// Sample function StartCT.
/// Starts the specified Container.
/// </summary>
/// <param name="ve_eid">Server ID of the Container.</param>
/// <returns>"OK" or error information.</returns>
public string StartCT(string ve_eid)
{
    try {

        // Instantiate the proxy class.
        vzaenvmBinding env =
(vzaenvmBinding)binder.InitBinding(typeof(vzaenvmBinding));

        // The main input object.
        start start_input = new start();

        // Set the Server ID of the Container.
        start_input.eid = ve_eid;

        // Start the VE.
        env.start(start_input);

        return "OK!";
    }
    catch (Exception e) {
        return "Exception: " + e.Message;
    }
}

```

Stopping and Restarting a VE is similar to the example above. The following two functions demonstrate how it's done.

```

/// <summary>
/// Sample function StopVE.
/// Stops a VE.
/// </summary>
/// <param name="ve_eid">Server ID of the container.</param>
/// <returns></returns>
public string StopVE(string ve_eid)
{
    try {
        vzaenvmBinding env =
(vzaenvmBinding)binder.InitBinding(typeof(vzaenvmBinding));
        stop1 stop_input = new stop1();

        // Set ID.
        stop_input.eid = ve_eid;

        // Stop the Container.
        env.stop(stop_input);

        return "OK!";
    }
    catch (Exception e) {
        return "Exception: " + e.Message;
    }
}

```

```

/// <summary>
/// Sample function RestartCT.
/// Restarts a Container.
/// </summary>
/// <param name="ve_eid">Server ID of the Container.</param>
/// <returns></returns>
public string RestartCT(string ve_eid)
{
    try {
        vzaenvmBinding env =
(vzaenvmBinding)binder.InitBinding(typeof(vzaenvmBinding));
        restart1 restart_input = new restart1();

        // Set ID.
        restart_input.eid = ve_eid;

        // Restart the Container.
        env.restart(restart_input);

        return "OK!";
    }
    catch (Exception e) {
        return "Exception: " + e.Message;
    }
}

```

Destroying a Container

To destroy a Container, use the `vzaenvmBinding.destroy()` method. The method accepts Server ID of a Container as a single parameter.

```

/// <summary>
/// Sample function DestroyCT.
/// Destroys a VE.
/// </summary>
/// <param name="ve_eid">Server ID of the Container.</param>
/// <returns>"OK" or error information.</returns>
public string DestroyCT(string ve_eid)
{
    try {
        vzaenvmBinding env =
(vzaenvmBinding)binder.InitBinding(typeof(vzaenvmBinding));
        destroy destroy_input = new destroy();

        // Set ID.
        destroy_input.eid = ve_eid;
        env.destroy(destroy_input);

        return "The Container has been destroyed.";
    }
    catch (Exception e) {
        return "Exception: " + e.Message;
    }
}

```

Suspending and Resuming a Container

The following two examples show how to suspend and then resume the operation of a Virtuozzo Container.

```

/// <summary>
/// Sample function SuspendCT.
/// Suspends a VE.
/// </summary>
/// <param name="ve_eid">Server ID of the Container.</param>
/// <returns>"OK" or error information.</returns>
public string SuspendCT(string ve_eid)
{
    try {
        vzaenvmBinding env =
(vzaenvmBinding)binder.InitBinding(typeof(vzaenvmBinding));
        suspend1 suspend_input = new suspend1();

        // Set Server ID.
        suspend_input.eid = ve_eid;

        // Suspend Container.
        env.suspend(suspend_input);

        return "OK!";
    }
    catch (Exception e) {
        return "Exception: " + e.Message;
    }
}

/// <summary>
/// Sample function ResumeVE.
/// Resumes a Container that was previously suspended.
/// </summary>
/// <param name="ve_eid">Server ID of the Container.</param>
/// <returns></returns>
public string ResumeVE(string ve_eid)
{
    try {
        vzaenvmBinding env =
(vzaenvmBinding)binder.InitBinding(typeof(vzaenvmBinding));
        resumel resume_input = new resumel();

        // Set Server ID.
        resume_input.eid = ve_eid;

        // Resume Container.
        env.resume(resume_input);

        return "OK!";
    }
    catch (Exception e) {
        return "Exception: " + e.Message;
    }
}

```

Getting Container Configuration Information

A Container configuration information is stored on the Hardware Node. This configuration (also called *virtual configuration*) is used by Virtuozzo to set the necessary Container parameters when the Container is started. To retrieve a Container configuration, use the `vzaenvmBinding.get_info` method. For the complete list and description of the input parameters, see the `vzaenvm/get_info` call in the *Parallels Agent XML Programmer's Reference* guide.

The following sample shows how to retrieve the complete configuration information for the specified Container.

```

/// <summary>
/// Sample function GetConfig.
/// Retrives Container configuration information.
/// </summary>
/// <param name="ve_eid">Server ID of the Container.</param>
/// <returns>
/// A string containing the Container configuration information.
/// </returns>
public string GetConfig(string ve_eid)
{
    string ve_info = "";
    try {
        // Instantiate the proxy class.
        vzaenvmBinding env =
(vzaenvmBinding)binder.InitBinding(typeof(vzaenvmBinding));

        // The input parameters.
        get_info2 getInfo_input = new get_info2();
        string[] eids = new string[1];

        // Set Server ID of the Container for which to get the info.
        eids[0] = ve_eid;
        getInfo_input.eid = eids;

        // Get the Container information from the Hardware Node.
        envType[] envtype = env.get_info(getInfo_input);

        // Get the Container configuration from the returned object.
        venv_configType veconfig = envtype[0].virtual_config;

        // Get Container name.
        ve_info += "Name: " + envtype[0].virtual_config.name + "\n";

        // Get Container description.
        if (envtype[0].virtual_config.description != null &&
envtype[0].virtual_config.description.Length != 0)
            ve_info += "Description: " +
System.Text.Encoding.ASCII.GetString(envtype[0].virtual_config.description) +
"\n" +

            //Get network configuration.
            "Network configuration: \n";
            if (envtype[0].virtual_config.address != null) {
                ve_info += "IP: " + veconfig.address[0].ip + "\n" +
                "Netmask: " + veconfig.address[0].netmask + "\n";
            }

        // Get Container hostname.
        ve_info += "HostName: " + veconfig.hostname + "\n" +
            // Get architecture
            "Architecture: " + veconfig.architecture + "\n" +

```

```

        // Get OS
        "OS name: " + veconfig.os.name + "\n" +
        "OS platform: " + veconfig.os.platform + "\n" +
        "OS kernel: " + veconfig.os.kernel + "\n" +
        "OS version: " + veconfig.os.version + "\n" +
        // Get status
        "Status: " + envtype[0].status.state.ToString() + "\n" +
        // Get QoS information.
        "QoS cur: " + veconfig.qos[0].cur.ToString() + "\n" +
        "QoS hard: " + veconfig.qos[0].hard.ToString() + "\n" +
        "QoS id: " + veconfig.qos[0].id + "\n" +
        "QoS soft: " + veconfig.qos[0].soft.ToString();// +"\n";

    }
    catch (Exception e) {
        ve_info += "Exception: " + e.Message;
    }
    return ve_info;
}

```

Configuring a Container

This section shows you how to modify a Container configuration. It is organized into subsections each demonstrating how to modify a particular configuration parameter. The basic idea behind modifying the Container configuration is simple. Agent SOAP API has classes that hold the Container configuration parameters. You instantiate the necessary classes (depending on the parameter type) and populate only those members (configuration parameters) that you would like to modify. You then submit the populated objects to Agent using the appropriate class and method. Upon receiving the new configuration, Agent will updated only those parameters that you specified in the input structure.

Modifying IP Address

Sample Function Parameters:

Name	Description
ve_eid	The Server ID of the Container for which you would like to modify the configuration info.
new_ip	The new IP address. A Virtuozzo Container may have multiple IP addresses assigned to it. When modifying the IP address information, all of the existing address information will be removed from the configuration and the new addresses will be put in their place. In this example, we will be operating with a single IP address for simplicity.
netmask	New netmask.
network	The name of the network interface for which you would like to modify the IP address settings.

Sample Function:

```

/// <summary>
/// Sample function ModifyIP.
/// Modifies the Container IP address.
/// </summary>
/// <param name="ve_eid">Server ID of the Container.</param>
/// <param name="new_ip">New IP address.</param>
/// <param name="netmask">New netmask.</param>
/// <param name="network">Network interface name.</param>
/// <returns>"OK" or error information.</returns>
public string ModifyIP(string ve_eid, string new_ip, string netmask, string
network)
{
    try {
        // Instantiate the proxy class.
        vzaenvmBinding env =
(vzaenvmBinding)binder.InitBinding(typeof(vzaenvmBinding));

        // The main input object.
        set2 set_input = new set2();

        // Set Server ID of the Container.
        set_input.eid = ve_eid;

        // The Container configuration structure.
        venv_configType1 veconfig = new venv_configType1();

        // Set ip addresses.
        ip_addressType[] ip_address = new ip_addressType[1];
        ip_address[0] = new ip_addressType();
        ip_address[0].ip = new_ip;
        ip_address[0].netmask = netmask;

        // The network interface information structure.
        net_vethType[] net = new net_vethType[1];
        net[0] = new net_vethType();

        // Set the network parameters.
        net[0].host_routed = new object();
        net[0].id = network;
        net[0].ip_address = ip_address;
        veconfig.net_device = net;
        set_input.config = veconfig;

        // Modify the Container configuration.
        env.set(set_input);

        return "OK!";
    }
    catch (Exception e) {
        return "Exception: " + e.Message;
    }
}

```

Modifying Hostname

```
/// <summary>
/// Sample function ModifyHostname.
/// Modifies Container hostname.
/// </summary>
/// <param name="ve_eid">Server ID of the Container.</param>
/// <param name="new_hostname">New hostname.</param>
/// <returns>OK/Error.</returns>
public string ModifyHostname(string ve_eid, string new_hostname)
{
    try {
        vzaenvmBinding env =
(vzaenvmBinding)binder.InitBinding(typeof(vzaenvmBinding));
        set2 set_input = new set2();

        // Set Server ID.
        set_input.eid = ve_eid;

        venv_configType1 veconf = new venv_configType1();

        // Set new hostname
        veconf.hostname = new_hostname;
        set_input.config = veconf;

        // Modify the Container configuration.
        env.set(set_input);

        return "OK!";
    }
    catch (Exception e) {
        return "Exception: " + e.Message;
    }
}
```

Modifying Container Name

```
/// <summary>
/// Sample function ModifyName.
/// Modifies Container name.
/// </summary>
/// <param name="ve_eid">Server ID of the Container.</param>
/// <param name="new_name">New Container name.</param>
/// <returns>OK/Error.</returns>
///
public string ModifyName(string ve_eid, string new_name)
{
    try {
        vzaenvmBinding env =
(vzaenvmBinding)binder.InitBinding(typeof(vzaenvmBinding));
        set2 set_input = new set2();

        // Set Server ID.
        set_input.eid = ve_eid;
        venv_configType1 veconf = new venv_configType1();

        // Set new Container name.
        veconf.name = new_name;
        set_input.config = veconf;

        // Modify the Container configuration.
        env.set(set_input);

        return "OK!";
    }
    catch (Exception e) {
        return "Exception: " + e.Message;
    }
}
```

Modifying QoS Settings

```
/// <summary>
/// Sample function ModifyQoS.
/// Modifies Container QoS settings.
/// </summary>
/// <param name="ve_eid">Server ID of the Container.</param>
/// <param name="qos_id">QoS ID.</param>
/// <param name="hard">New hard limit value.</param>
/// <param name="soft">New soft limit value.</param>
/// <returns></returns>
public string ModifyQoS(string ve_eid, string qos_id, int hard, int soft)
{
    try {
        vzaenvmBinding env =
(vzaenvmBinding)binder.InitBinding(typeof(vzaenvmBinding));
        set2 set_input = new set2();

        // Set Server ID.
        set_input.eid = ve_eid;

        venv_configType1 veconfig = new venv_configType1();

        // Set Container QoS.
        veconfig.qos = new qosType[1];
        veconfig.qos[0] = new qosType();

        // Set QoS ID.
        veconfig.qos[0].id = qos_id;

        // Set hard limit
        veconfig.qos[0].hardSpecified = true;
        veconfig.qos[0].hard = hard;

        // Set soft limit
        veconfig.qos[0].softSpecified = true;
        veconfig.qos[0].soft = soft;

        // Modify the Container configuration.
        set_input.config = veconfig;
        env.set(set_input);

        return "OK!";
    }
    catch (Exception e) {
        return "Exception: " + e.Message;
    }
}
```

Modifying DNS Server Assignment

```
/// <summary>
/// Sample function ModifyDNS.
/// Modifies Container DNS server assignment.
/// </summary>
/// <param name="ve_eid">Server ID of the Container.</param>
/// <param name="new_nameserver">New nameserver name.</param>
/// <returns>OK/Error.</returns>
public string ModifyDNS(string ve_eid, string new_nameserver)
{
    try {
        vzaenvmBinding env =
(vzaenvmBinding)binder.InitBinding(typeof(vzaenvmBinding));
        set2 set_input = new set2();

        // Set Server ID.
        set_input.eid = ve_eid;

        // Container configuration.
        venv_configType1 veconfig = new venv_configType1();

        // Network device.
        veconfig.net_device = new net_vethType[1];
        veconfig.net_device[0] = new net_vethType();

        // Set Container DNS.
        veconfig.net_device[0].nameserver = new string[1];
        veconfig.net_device[0].nameserver[0] = new_nameserver;

        // Modify Container configuration.
        set_input.config = veconfig;
        env.set(set_input);

        return "OK!";
    }
    catch (Exception e) {
        return "Exception: " + e.Message;
    }
}
```

Cloning a Virtuozzo Container

Cloning refers to a process of creating an exact copy (or multiple copies) of a Virtuozzo Container on the same Hardware Node. The new Container will have its own private area and root directories but the rest of the configuration parameters will be exactly the same. This means that even the parameters that should be unique for each individual Container (IP addresses, hostname, name) will be copied unchanged. You don't have an option to specify new values during the cloning operation. Instead, you will have to clone the Container first and then update the configuration of the new Container(s) in a separate procedure. There are a few exceptions to this rule. You can optionally specify custom private area and root directories for the new Container, but only if you are creating a single copy of the source Container. You also have an option to specify custom Container ID for each clone. If you don't want to set these options manually, their values will be selected automatically.

You can clone both running and stopped Containers. There are a few differences when cloning Containers on Windows and Linux platforms:



On Linux, running source Container will be suspended momentarily during the cloning operation. This is done in order to eliminate possible changes to the Container state and status. Once all the data is read from the source Container, the Container is resumed and the cloning operation proceeds normally.



On Windows, a snapshot of the source Container is taken on the fly, so the Container operation is never interrupted during cloning.

The following sample illustrates how to clone an existing Container. The name of the C# class that provides the cloning functionality is `relocatorBinding` (stepping ahead, this class also provides the Container migration functionality that we'll discuss in the following section). The XML API equivalent of the class is the `relocator` interface.

Sample Function Parameters:

Name	Description
<code>eid</code>	The Server ID of the Container to clone.
<code>count</code>	The number of clones to create.

Sample Function:

```
/// <summary>
/// Sample function CloneCT.
/// Create an exact copy of the specified Container.
/// </summary>
/// <param name="eid">The Server ID of the source Container.</param>
/// <param name="count">Number of copies to create.</param>
/// <returns>Server IDs of the new Virtuozzo Containers.</returns>
///
public string[] CloneCT(string eid, int count)
{
    cloneResponse response;

    try {
        // Instantiate the proxy class
        relocaterBinding relocater =
(relocaterBinding)binder.InitBinding(typeof(relocaterBinding));

        // The main input parameter.
        clone clone_input = new clone();

        // Set Server ID of the source Countainer.
        clone_input.eid = eid;

        // Number of copies to create.
        clone_input.count = 1;

        // Clone the Container(s).
        response = relocater.clone(clone_input);
    }
    catch (Exception e) {
        response = new cloneResponse();
        response.eid_list[0] = "Exception: " + e.Message;
        return response.eid_list;
    }
    return response.eid_list;
}
```

Migrating a Container to a Different Host

You can migrate an existing Container from one Hardware Node to another. The resulting Container is created as an exact copy of the source Container. To migrate a Container, the target Hardware Node must have Virtuozzo Containers software and Agent installed on it.

The following V2V (virtual-to-virtual) migration types are supported:

- *Offline migration.* Performed on a stopped or running source Container. If the Container is stopped, all its files are simply copied from the source host to the target host. If the Container is running, the files are first copied to the target machine and then the Container is stopped momentarily. At this point, the data that was copied to the target machine is compared to the original data and the files that have changed since the copying began are updated. The source Container is then started back up. The downtime depends on the size of the Container but should normally take only a minute or so. Offline migration is the default migration type.
- *Simple online migration.* Performed on a running source Container. In the beginning of the migration process, the Container becomes momentarily locked and all of its data, including the states of all running processes, is dumped into an image file. After that, the Container operation is resumed, and the dump file is transferred to the target computer where Virtuozzo Containers automatically creates a new Container from it.
- *Lazy online migration.* Instead of migrating all of the data in one big step (as in simple online migration above), lazy migration copies the data over a time period. Initially, only the data that is absolutely necessary to bring the new Container up is copied to the target host. The rest of the data remains locked on the source host and is copied to the destination host on as-needed basis. By using this approach, you can decrease the services downtime to near zero.
- *Iterative online.* During the iterative online migration, the Container memory is transferred to the destination node before the Container data is dumped into an image file. Using this type of online migration allows to attain the smallest service delay.
- *Iterative + lazy online migration.* This type of online migration combines the techniques used in both the lazy and iterative migration types, i.e. some part of Container memory is transferred to the destination host before dumping a Container, and the rest of the data is transferred on-demand after the Container has been successfully created on the target host.

On successful migration, the original Container will no longer exist on the source node. This is done in order to avoid possible conflicts that may occur if both Containers -- the original and the copy -- are running at the same time. Although the original Container will no longer show up in the Container list on the source node, the Container data will not be deleted. By default, the data is kept in its original location (the Container private area) but the private area directory itself will be renamed. If you wish, you can completely remove the original Container data from the source node by including the `options/remove` parameter in the request.

The name of the C# class that provides the migration functionality is `relocatorBinding`. The XML API equivalent is the `relocator` interface.

The following sample shows how to perform a V2V migration.

Sample Function Parameters:

Name	Description

eid	The Server ID of the source Container.
mn_type	Migration type: 0 -- Offline 1 -- Simple online 2 -- Lazy online 3 -- Iterative online 4 -- Iterative lazy online
ip_address	This and the rest of the parameters are the connection and login information that will be used to log in to the target Hardware Node. The target Hardware Node IP address.
port	Port number.
protocol	Communication protocol to use: SSL -- SSL over TCP/IP. TCP -- plain TCP/IP. NamedPipe -- named pipe.
username	User name. The user must have sufficient rights to connect to the target Hardware Node.
realm	Realm ID. The ID of the authentication database against which to authenticate the specified user. In this example, we will be using the <code>system</code> Realm -- the host operating system user registry.
password	User password.

Sample Function:

```

/// <summary>
/// Sample function Migrate.
/// Migrates a Container to a different Hardware Node.
/// </summary>
/// <param name="eid">The Server ID of the source Container.</param>
/// <param name="mn_type">Migration type.</param>
/// <param name="ip_address">Target HN IP address.</param>
/// <param name="port">Target HN port number.</param>
/// <param name="protocol">Communication protocol.</param>
/// <param name="username">
/// User name with which to login to the
/// target HN.
/// </param>
/// <param name="realm">
/// Realm ID on the target HN against which to authenticate the user.
/// </param>
/// <param name="password">User password.</param>
/// <returns>"OK" or error information.</returns>
///
public string Migrate(string eid, int migration_type, string ip_address, uint
port, string protocol, string username, string realm, string password)
{
    try {
        relocaterBinding relocater =
(relocaterBinding)binder.InitBinding(typeof(relocaterBinding));
        migrate_v2v v2v_input = new migrate_v2v();

        // Set Server ID of the source Container.
        v2v_input.eid_list = new string[1];
        v2v_input.eid_list[0] = eid;

        /* Set migration type.
        * The "options" member allows you to set other
        * migration options. See Agent XML Reference
        * for more info.
        */
        v2v_input.options = new v2v_migrate_optionsType();
        v2v_input.options.type = migration_type;

        // Set the target Nardware Node connection info.
        v2v_input.dst = new connection_infoType();
        connection_infoType connection_parm =
(connection_infoType)v2v_input.dst;

        // Set the target Node IP address.
        v2v_input.dst.address = ip_address;

        // Set the port number.
        v2v_input.dst.portSpecified = true;
        v2v_input.dst.port = port;

        // Set protocol.
        v2v_input.dst.protocol = protocol;

        // Set login parameters.
        v2v_input.dst.login = new auth_nameType();
        v2v_input.dst.login.name =
System.Text.ASCIIEncoding.ASCII.GetBytes(username);
        v2v_input.dst.login.realm = realm;

        // Set user password.
        v2v_input.dst.password =
System.Text.ASCIIEncoding.ASCII.GetBytes(password);

        // Set infinite timeout for the request.

```

```

relocator.Timeout = -1;
relocator.migrate_v2v(v2v_input);

return "OK";
}
catch (Exception e) {
return "Exception: " + e.Message;
}
}

```

Backup Operations

Agent SOAP API provides a set of methods that allow to perform Virtuozzo Container backup and restore operations. The following subsections describe some of the most common operation scenarios and provide code samples.

Backing up a Container

The following sample illustrates how to use the `backupmBinding.backup_env` method to create a backup of a Virtuozzo Container.

Sample Function Parameters:

Name	Description
<code>eid</code>	The Server ID of the source Container.
<code>description</code>	Backup description.
<code>type</code>	Backup type: 0 -- Full (default). A full backup is a starting point for all other backup types. 1 -- Incremental. Only the files that have changed since the last full, incremental, or differential backup are included. When restoring from an incremental backup, you'll need the latest full backup as well as every incremental and/or differential backup that you've made since the last full backup. 2 -- Differential. Only the files that have changed since the last full backup are included. When restoring from a differential backup, only the latest differential backup itself and the latest full backup is needed.
<code>compression</code>	Compression level: 0 -- no compression. 1 -- normal (default). 2 -- high. 3 -- maximum.

	<p><i>The following parameters are used to specify the backup server connection and login information. The sample function provided here illustrates how to specify the backup server connection and login information manually.</i></p> <p>Note: To use a remote computer as a backup server, you must install Virtuozzo Containers software on it.</p>
ip	Backup server IP address.
user	Login name.
password	Login password.
realm	<p>Realm ID against which to authenticate the user. The Realm definition must exist in the Agent configuration on the backup server.</p> <p>On a fresh Agent installation, the only Realm available is <code>System</code> -- a predefined Realm that refers to the operating system user registry on the Hardware Node. The System Realm ID is <code>00000000-0000-0000-0000-000000000000</code>.</p>
protocol	<p>Communication protocol:</p> <p>SSL -- SSL over TCP/IP.</p> <p>TCP -- plain TCP/IP.</p> <p>NamedPipe -- named pipe.</p>
port	<p>Port number. Agent on the source server will be connecting to the Agent on the backup server, the TCP port numbers therefore are the standard Agent options:</p> <p>4433 -- TCP/IP connection.</p> <p>4434 -- SSL over TCP/IP connection.</p>

Sample Function:

```

/// <summary>
/// Sample function BackupVE.
/// Performs a Container backup.
/// </summary>
/// <param name="eid">Server ID of the source Container.</param>
/// <param name="description">
/// User-defined backup description.
/// </param>
/// <param name="type">Backup type.</param>
/// <param name="compression">Compression level.</param>
/// <param name="ip">Target HN IP address.</param>
/// <param name="user">
/// User name with which to login to
/// the target Nardware Node.
/// </param>
/// <param name="password">User password.</param>
/// <param name="realm">Realm ID.</param>
/// <param name="protocol">Communication protocol.</param>
/// <param name="port">Target HN port number</param>
/// <returns>Backup ID.</returns>
///
public string BackupVE(string eid, string description, int type, int
compression, string ip, string user, string password, string realm, string
domain, string protocol, uint port)
{
    try {
        // Instantiate the proxy class
        backupmBinding backupm =
(backupmBinding)binder.InitBinding(typeof(backupmBinding));

        // The main input object.
        backup_env backup_input = new backup_env();

        // Set Server ID of the Container.
        backup_input.env_list = new string[1];
        backup_input.env_list[0] = eid;

        // Set backup options.
        backup_optionsType options = new backup_optionsType();

        // Backup description.
        options.description =
System.Text.ASCIIEncoding.ASCII.GetBytes(description);

        // Backup type.
        options.typeSpecified = true;
        options.type = type;

        // Compression level.
        options.compressionSpecified = true;
        options.compression = compression;

        // Set the backup server login information.
        auth_nameType login_info = new auth_nameType();

        // User name.
        login_info.name = System.Text.ASCIIEncoding.ASCII.GetBytes(user);

        // Realm ID.
        login_info.realm = realm;

        // Domain name.
        login_info.domain = System.Text.ASCIIEncoding.ASCII.GetBytes(domain);

        // Set the backup server connection information.

```

```

        connection_infoType connection = new connection_infoType();

        // Backup server IP address.
        connection.address = ip;

        // Communication protocol.
        connection.protocol = protocol;

        // Port number.
        connection.portSpecified = true;
        connection.port = port;

        // Password.
        connection.password =
System.Text.ASCIIEncoding.ASCII.GetBytes(password);

        // Finalize the input.
        connection.login = login_info;
        backup_input.backup_server = connection;
        backup_input.backup_options = options;

        // Set infinite timeout for the request.
        backupm.Timeout = -1;

        // Start backup.
        return backupm.backup_env(backup_input)[0].id;
    }
    catch (Exception e) {
        return "Exception: " + e.Message;
    }
}

```

The function invocation example:

```

createBackup( "ac57a9c3-573b-481a-b398-d2fb0467cf4b", "my backup", 0, 0,
"10.16.3.80", "root", "my_password", "00000000-0000-0000-0000-000000000000",
"TCP", 4433 );

```

Listing Backups

To get the list of the existing backups, use the `backupmBinding.list` method. The method retrieves the list of backups from the Hardware Node. If your backup archives are located on a remote backup server, you have to establish a direct Agent connection with it and execute the call there. If you have a Virtuozzo group set up, you can use this method on the Master Node to get the list of backups from any Slave Node in the group.

The `backupmBinding.list` method returns the following backup information:

Name	Description
time	Backup date and time.
size	The size of the backup archive.
type	Backup type: 0 -- Full. 1 -- Incremental. 2 -- Differential.

id	Backup ID. The ID is assigned to a backup by Agent when the backup is created. The backup ID is universally unique. You have to obtain the backup ID in order to restore a Container from a backup.
storage_eid	Server ID of the Node where the backup archive is located.
info	Additional backup information.
eid	The original Server ID of the source Container.
description	Backup description. An optional property set by user. May be empty.
count	The total number of backups stored on this Node (<code>storage_eid</code>) for this Container (<code>eid</code>).
capability	Backup capabilities. Specifies miscellaneous backup properties.

The following sample illustrates how to use the `backupmBinding.list` method to get the list of all of the available backups from the Hardware Node.

Sample Function:

```

/// <summary>
/// Sample function LisBackups.
/// Retrieves the list of backups from the
/// Hardware Node.
/// </summary>
/// <returns>
/// A string containing the list of backups
/// with detailed information for each backup.
/// </returns>
///
public string ListBackups()
{
    string list_result = "";
    try {
        // Instantiate the proxy class
        backupmBinding backupm =
(backupmBinding)binder.InitBinding(typeof(backupmBinding));

        // The main input parameter.
        list7 list_input = new list7();

        // Get the list of backups.
        backupType[] backups = backupm.list(list_input);

        // Iterate through the returned backup structure and
        // populate a string variable with the results.
        if (backups.Length != 0) {
            foreach (backupType backup in backups) {
                if (backup.description != null) {
                    list_result += "\nDescription: ";

                    list_result +=
System.Text.ASCIIEncoding.ASCII.GetString(backup.description) +
                    "\nID: " + backup.id +
                    "\nServer ID: " + backup.eid +
                    "\nCount: " + backup.count.ToString() +
                    "\nSize: " + backup.size.ToString() +
                    "\nType: " + backup.type.ToString() +
                    "\nInfo name: " + backup.info.name +
                    "\nTime: " + backup.time.ToString();
                }
            }
        }
        else {
            list_result += "\nNo backups found.";
        }
    }
    catch (Exception e) {
        list_result += "Exception: " + e.Message;
    }

    return list_result;
}

```

The following example illustrates how to use the optional parameters in the backup listing call. The sample function retrieves the most recent backup for the specified Virtuozzo Container from the specified Slave Node in a Virtuozzo group.

Sample Function Parameters:

Name	Description
slave_eid	The Server ID of the Slave Node where the backups are stored. This can be any Slave Node in a Virtuozzo group. To get the Server IDs of the Slave Nodes, use the <code>server_groupBinding.get_list</code> method.
ve_eid	The Server ID of the Virtuozzo Container to get the backup info for. The Container can reside on any Node in the Virtuozzo group.
latest	A flag indicating whether to retrieve the information about the most recent backup or about all of the available backups.

Sample Function:

```

/// <summary>
/// Sample function ListBackupsVZgroup.
/// </summary>
/// <param name="slave_eid">
/// Server ID of a slave Node from which to get the list of backups.
/// </param>
/// <param name="ve_eid">
/// Server ID of the Container to search for in the backups.
/// </param>
/// <param name="latest">
/// A flag. If set to true, the function will retrieve the
/// information about the latest available backup only.
/// If set to false, all available backups matching the
/// specified criteria will be retrieved.
/// </param>
/// <returns>A string containing the backup informaiton.</returns>
///
public string ListBackupsVZgroup(string slave_eid, string ve_eid, Boolean
latest)
{
    string list_result = "";
    try {
        // Instantiate the proxy class
        backupmBinding backupm =
        (backupmBinding)binder.InitBinding(typeof(backupmBinding));

        // The main input object.
        list7 list_input = new list7();

        // Set slave Node's Server ID.
        list_input.options.storage_eid = slave_eid;

        // Set Container's Server ID.
        list_input.options.eid = ve_eid;

        // Set the "latest" flag.
        if (latest) {
            list_input.options.latest = new Object();
        }

        // Get the backup info.
        backupType[] backups = backupm.list(list_input);

        // Iterate through the returned backup array.
        if (backups.Length != 0) {
            foreach (backupType backup in backups) {
                if (backup.description != null) {
                    list_result += "\nDescription: ";

                    list_result +=
System.Text.AsciiEncoding.Ascii.GetString(backup.description) +
                    "\nID: " + backup.id +
                    "\nServer ID: " + backup.eid +
                    "\nCount: " + backup.count.ToString() +
                    "\nSize: " + backup.size.ToString() +
                    "\nType: " + backup.type.ToString() +
                    "\nInfo name: " + backup.info.name +
                    "\nTime: " + backup.time.ToString();
                }
            }
        }
        else {
            list_result += "\nNo backups found.";
        }
    }
}

```

```
    catch (Exception e) {  
        list_result += "Exception: " + e.Message;  
    }  
    return list_result;  
}
```

Restoring a Container

To restore a Container from a backup archive, you first have to obtain its ID. The backup that you are restoring from must be one of the following:

- A full backup, containing all the files and directories that are required for the Container to operate properly.
- An incremental backup, plus all the prior incremental and differential backups, and the original full backup from the same sequence.
- A differential backup, plus the original full backup from the same sequence.

By default, the Container will be restored on the Hardware Node that you are currently connected to. When you are restoring a Container in a Virtuozzo group, an attempt will be made to restore it to the original Node. If the original Node is no longer registered with the group, you'll have to set the target Node manually or the operation will fail. Regardless of the conditions under which the restore operation is performed, the resulting Container will always have the same Server ID as the original Container.

The samples in this section illustrates how to use the `backupmBinding.restore_env` method to restore a Container from a backup archive. The following sample function accepts a local backup ID and restores a Container on the current Hardware Node.

Sample Function:

```

/// <summary>
/// Sample function RestoreBackup.
/// Restores a Container from a backup.
/// </summary>
/// <param name="backup_id">Backup ID.</param>
/// <returns>"OK" or error information.</returns>
///
public string RestoreBackup(string backup_id) {
    string list_result = "";
    try {
        backupmBinding backupm =
        (backupmBinding)binder.InitBinding(typeof(backupmBinding));

        // The main input object.
        restore_env restore = new restore_env();

        // Set backup id.
        restore.backup_id = backup_id;

        // Start restore.
        backupm.restore_env(restore);

        return "OK!";
    }
    catch (Exception e) {
        list_result += "Exception: " + e.Message;
    }
    return list_result;
}

```

The following sample shows how to restore a Container from a remote backup. In it, we pass the backup ID and the backup server connection and login information.

Sample Function Parameters:

Name	Description
backup_id	Backup ID.
	<i>The rest of the parameters specify the remote backup server connection and login information.</i>
ip	IP address.
user	User name.
realm	Realm ID.
port	TCP port number.
protocol	Communication protocol.
password	User password.

Sample Function:

```

/// <summary>
/// Sample function RestoreRemoteBackup.
/// Restores a Container from a remotely stored backup.
/// </summary>
/// <param name="backup_id">Backup ID.</param>
/// <param name="ip">Remote HN IP address.</param>
/// <param name="user">
/// User name with which to login to the remote HN node.
/// </param>
/// <param name="realm">Realm ID.</param>
/// <param name="port">Port number.</param>
/// <param name="protocol">Communication protocol.</param>
/// <param name="password">Password.</param>
/// <returns>"OK" or error information.</returns>
///
public string RestoreRemoteBackup(string backup_id, string ip, string user,
string realm, uint port, string protocol, string password)
{
    string list_result = "";

    try {
        // Instantiate the proxy class
        backupmBinding backupm =
(backupmBinding)binder.InitBinding(typeof(backupmBinding));

        // The main input object.
        restore_env restore = new restore_env();

        //Set backup id.
        restore.backup_id = backup_id;

        // The backup server connection info.
        connection_infoType connection = new connection_infoType();

        // Set IP address.
        connection.address = ip;
        auth_nameType name = new auth_nameType();

        // Set user name.
        name.name = System.Text.ASCIIEncoding.ASCII.GetBytes(user);

        // Set Realm ID.
        name.realm = realm;

        // Set port number.
        connection.portSpecified = true;
        connection.port = port;

        // Set communication protocol name.
        connection.protocol = protocol;
        connection.login = name;

        // Set user password.
        connection.password =
System.Text.ASCIIEncoding.ASCII.GetBytes(password);

        // Finalize the backup server connection and login
        // parameters.
        restore.backup_server = connection;

        // Start restore.
        backupm.restore_env(restore);

        return "OK!";
    }
}

```

```
catch (Exception e) {  
    list_result += "Exception: " + e.Message;  
}  
return list_result;  
}
```

The function invocation example:

```
restoreBackup("85a2dd71-9133-7c44-8521-f6bd517f17ca/0", "10.16.3.80", "root",  
"my_password", "00000000-0000-0000-0000-000000000000", "TCP", 4433);
```

Getting Container Information From a Backup

Use the `backupmBinding.get_info` method to get the detailed Container information from the specified backup archive. The information that can be retrieved includes the basic Container information (Server ID, status, host Server ID) and the complete Container configuration.

On a standalone Hardware Node, the method can access only the local backups. To get the info about a backup on a remote backup server, you'll have to establish a direct Agent connection with it. In a Virtuozzo group, the method can get information about a backup located on any Slave Node.

The following sample shows how to retrieve the Container information from a backup.

Sample Function:

```

/// <summary>
/// Sample function GetInfo.
/// Retrives information about the Container stored in the
/// specified backup.
/// </summary>
/// <param name="backup_id">Backup ID.</param>
/// <returns>Container information.</returns>
///
public string GetInfo(string backup_id)
{
    string info_result = "";

    try {

        // Instantiate the proxy class
        backupmBinding backupm =
(backupmBinding)binder.InitBinding(typeof(backupmBinding));

        // The main input object.
        get_info3 info = new get_info3();

        // Set backup id.
        info.backup_id = backup_id;

        // Populate the retrieval criteria.
        get_env_info_optionsType options = new get_env_info_optionsType();

        /* The options.env option is used as a flag.
        * If the options.env object is created, the result will
        * contain the complete Container info, including the Container
configuration.
        * If not, only the basic Container info will be included.
        */
        options.env = new get_env_info_optionsTypeEnv();

        /* If options.excludes object is created,
        * the result will also contain the names of
        * the Container files and directories that were included in
        * or excluded from the backup.
        */
        options.excludes = new object();

        // Set the options.
        info.options = options;

        // Get the Container info.

```

```
env_backup_dataType data =
(env_backup_dataType)backupm.get_info(info).info;

// Populate a string variable with the returned data.
info_result += "\nEnvironment info:" +
    "\n  Eid: " + data.env.eid +
    "\n  Parent eid: " + data.env.parent_eid +
    "\n  Name: " + data.env.virtual_config.name;

// Get the configuration description.
if (data.env.virtual_config.description != null &&
    data.env.virtual_config.description.Length != 0) {
    info_result += "\n  Description: " +
System.Text.ASCIIEncoding.ASCII.GetString(data.env.virtual_config.description)
;
    }

// Get the OS info.
info_result += "\n  Status: " + data.env.status.state.ToString()+
    "\n  Architecture: " + data.env.virtual_config.architecture+
    "\n  OS name: " + data.env.virtual_config.os.name+
    "\n  OS platform: " + data.env.virtual_config.os.platform+
    "\n  OS version: " + data.env.virtual_config.os.version+
    "\n  OS kernel: " + data.env.virtual_config.os.kernel;

// Get the Container IP address.
if (data.env.virtual_config.address != null) {
    info_result += "\n  IP: " + data.env.virtual_config.address[0].ip;
}
else {
    info_result += "\n  No IP address!";
}

// Get the hostname.
if (data.env.virtual_config.hostname != null) {
    info_result += "\n  Hostname: " +
data.env.virtual_config.hostname;
}

// Get the sample configuration ID.
if (data.env.virtual_config.base_sample_id != null) {
    info_result += "\n  Base sample id: " +
data.env.virtual_config.base_sample_id;
}
}
catch (Exception e) {
    info_result += "Exception: " + e.Message;
}
return info_result;
}
```

Performance Monitor

Performance Monitor is an operator that allows to monitor the performance of the Hardware Node and Virtuozzo Containers. By monitoring the utilization of the system resources, you can acquire an important information about your Virtuozzo system health. Performance Monitor can track a range of processes in real time and provide you with the results that can be used to identify current and potential problems. It can assist you with the tracking of the processes that need to be optimized, monitoring the results of the configuration changes, identifying the resource usage bottlenecks, and planning of upgrades.

Agent SOAP API provides the `perf_monBinding` class that allows to retrieve performance reports from the Hardware Node. The types of reports include the performance of the Hardware Node itself and the performances of individual Virtuozzo Containers. You can select the type and a particular aspect of the server performance that you would like to see. This performance type is called a *class*. The performance aspect is called a *counter*. The following sub-section describes classes and counters in more detail.

Classes, Instances, Counters

Performance Class

Performance class is a type of the system resource that can be monitored. This includes CPU, memory, disk, network, etc. A class is identified by ID. See [Appendix A: Performance Counters](#) (p. 150) for a complete list of classes. Please note that there are two separate groups of classes: one is used for monitoring Virtuozzo Containers and the other for monitoring Hardware Nodes.

Class Instance

While class identifies the type of the system resource, the term "instance" refers to a particular device when multiple devices of the same type exist in the system. For example, network in general is a class, but each network card installed in the system is an instance of that class. Each class has at least one instance, but not all classes may have multiple instances. An instance is identified by a name assigned to the corresponding device by the operating system or Virtuozzo Containers. [Appendix A: Performance Counters](#) (p. 150) provides information on how to obtain a list of instances for each class.

Performance Counter

Counters are used to measure various aspects of a performance, such as the CPU times, network rates, disk usage, etc. Each class has its own set of counters. Counter data is comprised of the current, minimum, maximum, and average values. For the complete list of counters see [Appendix A: Performance Counters](#) (p. 150).

Getting a Performance Report

The following example consists of two functions working together. The functions retrieve the latest performance report using the specified Server ID, performance class, and performance counter. The `GetPerfData` function initializes and populates the necessary input parameters, gets the performance data from Agent, and then calls the `getData` function (described below) that extracts the data and puts it into a string that can be displayed on the screen.

Sample Function Parameters:

Name	Description
<code>eid</code>	Server ID of the Container for which to retrieve the performance data.
<code>class_name</code>	The name of the performance class.
<code>counter_name</code>	The name of the performance counter.

Sample Function:

```

/// <summary>
/// Sample function GetPerfData.
/// Gets the Container or the Hardware Node performance data.
/// </summary>
/// <param name="eid"></param>
/// <param name="class_name"></param>
/// <param name="counter_name"></param>
/// <returns>A string containing the performance data.</returns>
///
public string GetPerfData(string eid, string class_name, string counter_name,
string class_instance)
{
    string perf_data = "";
    try {

        // Create binding object.
        perf_monBinding perf_mon =
(perf_monBinding)binder.InitBinding(typeof(perf_monBinding));

        // The main input object.
        get5 get_input = new get5();

        // Set Server ID.
        get_input.eid_list = new string[1];
        get_input.eid_list[0] = eid;

        /* Set the performance class name.
        * Multiple classes can be set if desired.
        */
        get_input.@class = new classType1[1];
        get_input.@class[0] = new classType1();
        get_input.@class[0].name = class_name;

        // Set class instance.
        get_input.@class[0].instance = new classTypeInstance[1];
        get_input.@class[0].instance[0] = new classTypeInstance();
        if (class_instance.Length != 0) {
            get_input.@class[0].instance[0].name = class_instance;
        }

        // Set counter. Multiple counters can be set if desired.
        get_input.@class[0].instance[0].counter = new string[1];
        get_input.@class[0].instance[0].counter[0] = counter_name;

        /* Get the performance data. The returned data is
        * extracted using the GetData helper function, which
        * is defined below.
        */
        GetData(perf_mon.get(get_input), out perf_data);
    }
    catch (Exception e) {
        perf_data += "Exception: " + e.Message;
    }
    return perf_data;
}

/// <summary>
/// Sample function GetData.
/// This is a helper function that extracts the performance
/// data retrieved by the getPerfData function defined above.
/// </summary>
/// <param name="counters_dat">
/// Contains the data for each class, instance, and counter that
/// were specified in the request that returned this object (the

```

```

/// perf_mon.get() call above). To extract the data, we have to iterate
through all
/// of them.
/// </param>
/// <param name="counters_info">
/// Output. A string containing the extracted data.
/// </param>
///
public void GetData(perf_dataType[] counters_dat, out string counters_info)
{
    counters_info = "";
    if (counters_dat.Length != 0) {
        foreach (perf_dataType counter_dat in counters_dat) {
            if (counter_dat.@class != null) {
                foreach (perf_dataTypeClass dat in counter_dat.@class) {
                    counters_info += "\n Class name: " + dat.name + "\n" +
                    "Instances:\n";
                    if (dat.instance != null) {
                        foreach (perf_dataTypeClassInstance instance in
dat.instance) {
                            counters_info += " DataClassInstance: " +
instance.name + "\n";
                            if (instance.counter != null) {
                                foreach (perf_dataTypeClassInstanceCounter
counter in instance.counter) {
                                    counters_info += "    \nName:" +
counter.name + "\n" +
+
+
+
+
                                    "        avg: " + counter.value.avg + "\n"
                                    "        cur: " + counter.value.cur + "\n"
                                    "        max: " + counter.value.max + "\n"
                                    "        min: " + counter.value.min;
                                }
                            }
                            else {
                                counters_info += " No counters." + "\n";
                            }
                        }
                    }
                    else {
                        counters_info += "No instances." + "\n";
                    }
                }
            }
            else {
                counters_info += "No classes." + "\n";
            }

            counters_info += "Intervals:\n" +
            "Start time: " + counter_dat.interval.start_time + "\n" +
            "End time: " + counter_dat.interval.end_time + "\n" +
            "Server ID: " + counter_dat.eid + "\n";
        }
    }
    else {
        counters_info += "No data returned.";
    }
}

```

Monitoring Alerts

Alerts are notifications that report the system resource allocation problems such as approaching or exceeding certain limits. Alerts are usually used for monitoring of the Container health, predicting its performance, or collecting information that can be used to optimize the Container performance. Use the `AlertmBinding` class to check if a Container has alerts of any kind currently raised and to retrieve the alert data if it does.

The alert levels are described in the table below.

Alert level	ID	Description
Green	0	Normal operation. This alert is raised when one of the higher-level alerts is canceled.
Yellow	1	Moderately dangerous situation. The specified parameter is coming close (within 10%) to its soft limit barrier.
Red	2	Critical situation. The parameter exceeded its soft limit or came very close to the hard limit. Depending on the parameter type, either some process can be killed at any time now, or the next resource allocation request can be refused.

A Virtuozzo Container may have multiple alerts raised at any given time. The following function demonstrates how you can check if a Container has any alerts currently raised, and to retrieve the alert information if it does. The function accepts the list of Containers for which to check and retrieve the alert information.

```

/// <summary>
/// Sample function GetAlerts.
/// Retrieves the system alert information for the specified Container.
/// </summary>
/// <param name="ve_eid">
/// Server ID of the Container to get the alerts for.
/// </param>
/// <returns>A string containing the alert information.</returns>
///
public string GetAlerts(string[] ve_eid)
{
    string list_result = "";
    try {
        // Instantiate the proxy class
        alertmBinding alertm =
(alertmBinding)binder.InitBinding(typeof(alertmBinding));

        // The main input object.
        get_alerts get_alerts_input = new get_alerts();

        //Set Container list.
        get_alerts_input.eid_list = ve_eid;

        // Get the alert information.
        foreach (eventType al_event in alertm.get_alerts(get_alerts_input)) {
            list_result += "Data: \n";

            // Get the alert data.
            resource_alertType res_data =
(resource_alertType)al_event.data.event_data;
            // Read the alert data.
            list_result += " Class: " + res_data.@class + "\n" +
                // Get counter.
                " Counter: " + res_data.counter + "\n" +
                // Get eid.
                " Eid: " + res_data.eid + "\n" +
                // Get instance.
                " Instance: " + res_data.instance + "\n" +
                // Get type.
                " Type: " + res_data.type.ToString() + "\n" +
                // Get current value.
                " Cur: " + res_data.cur + "\n" +
                // Get hard limit.
                " Hard: " + res_data.hard + "\n" +
                // Get soft limit.
                " Soft: " + res_data.soft + "\n" +
                // Get event name.
                "Name: " + al_event.info.name + "\n" +
                // Get count.
                "Count: " + al_event.count.ToString() + "\n" +
                // Get event category.
                "Category: " + al_event.category + "\n" +
                // Get event message.
                "Message: " +
System.Text.AsciiEncoding.ASCII.GetString(al_event.info.message) + "\n" +
                // Get parameters
                "Parameters: ";
            /* Call the helper function to extract the
            * event message parameter values.
            */
            GetParams(al_event.info.parameter, ref list_result);
        }
    }
}

```

```
    }
  }
  catch (Exception e) {
    list_result += "Exception: " + e.Message;
  }
  return list_result;
}

/// <summary>
/// Sample function GetParams.
/// This is a helper function that extracts the
/// alert message parameter values.
/// </summary>
/// <param name="parameter">The name of the parameter.</param>
/// <param name="list">
/// Output. Values.
/// </param>
///
void GetParams(infoType[] parameter, ref string list)
{
  string ss = " ";

  foreach (infoType param in parameter) {
    list += ss + "Message: " +
System.Text.ASCIIEncoding.ASCII.GetString(param.message) +
    "   Info name: " + param.name + "\n";

    if (param.parameter != null) {
      GetParams(param.parameter, ref list);
    }
  }
}
```

Other SOAP Clients and Their Known Issues

Visual Basic .NET

Microsoft .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 `Agent 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.ap  
ache.axis.components.net.SunFakeTrustSocketFactory");
```

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

Troubleshooting

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, Agent 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, Agent responded with error
Details:
1004
Error invoking vzctl utility: Container is already running
```

Solution: Check the state of your Container that is used for the current operation. In the example above, you try to start a Container and get the error message. This kind of error may occur when you are starting a Container 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 servicel = new MyService();

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

// The timeout is set to 10 minutes.
servicel.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

In This Chapter

Agent Configuration	145
Internal Request Scheduler	145

Agent Configuration

Agent configuration consists of a set of configuration parameters for each of the Agent operators. The configuration information is stored in a file as an XML document. On Agent 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 Agent from a client program just like any other request. Agent, 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. Agent configuration information can be retrieved and modified using Agent API.

Internal Request Scheduler

In order to be able to process more requests and decrease the load on the Hardware Node, VZAgent features a simple internal request scheduler. This section describes the scheduler internals and how you can take advantage of its functionality in your client applications.

Message Classification and Priorities

The messages traveling through Agent 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 Agent before their processing by the operator connection) are not the same as the priorities of the internal Agent messages. The former are translated into the latter by operator connection, i.e. on their entrance to Agent.

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 a server, 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 Virtuozzo Containers, cloning, migration, template installation, and others.

Emergency messages are for internal Agent 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 server management operator can simultaneously serve up to 4 Virtuozzo Container creation processes, up to 10 Container stops, and up to 20 Container configuration fetches by default. It means that the director forks another server 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 its processing begins when one of the existing operators becomes available.

Pools are strictly concerned with a particular operator and don't intersect in any way. As an example, the computer management operator pool never interferes with the server 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 "available" to "busy". Upon completion, the status of the operator is switched to "available" 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 Agent 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 = $+1/\text{comeback_ratio}$.

Normal Dynamic Limit = $+1/\text{comeback_ratio}/(20/10)$.

Heavy Dynamic Limit = $+1/\text{comeback_ratio}/(20/4)$

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.

CHAPTER 6

Appendix A: Performance Counters

Performance Classes

There are two groups of performance classes: one is for monitoring Virtuozzo Containers, and the other is for monitoring the host server (Hardware Node). Both groups are listed in the following table.

Class ID	Resource Type	Class Instances
Container classes		
counters_vz_cpu	CPU	N/A
counters_vz_ubc	UBC (User Bean Counters).	N/A
counters_vz_net	Container network.	Use <code>vznetstat</code> command-line utility to obtain a list of instances. The <code>Net.Class</code> column will contain the available instances. The rows with <code>CTID = 0</code> (Container 0 or Hardware Node) are not applicable.
counters_vz_quota	Disk quota.	N/A
counters_vz_loadavg	Load average.	N/A
counters_vz_system	System info.	N/A
counters_vz_slm	SLM	N/A
counters_vz_memory	Memory	N/A
counters_vz_hw_net	Hardware Node network.	Use <code>vznetstat</code> command-line utility to obtain a list of instances. The <code>Net.Class</code> column will contain the available instances. Only the rows with <code>CTID=0</code> (Container 0 or Hardware Node) must be looked at.
Hardware Node classes		
counters_cpu	CPU	N/A
counters_disk	Disk	The name of the hard disk device.
counters_memory	Memory	N/A
counters_net	Network	The name of the network interface.
counters_loadavg	CPU	N/A
counters_system	System info.	N/A

Performance Counters

Note: UBC failcounters are not supported in the current version of Parallels Agent.

The tables below contain lists of performance counters by their parent class. The table columns are:

Counter ID	Counter ID. The IDs are used in Agent calls as input/output parameters.
Value	The data type of the counter value(s).
Type	Counter type. A performance counter may be one of the following types: <ul style="list-style-type: none"> ▪ <i>Periodic counter (type 0)</i>. Contains the minimum, maximum, and average values for the given time period. ▪ <i>Incremental counter (type 1)</i>. The value of an incremental counter is always higher or equals to the previous value. A good example is a network counter that counts the number of bytes the interface has sent or received. The minimum, maximum, and average values are the same and represent the difference between the current value and the value from the previous report. ▪ <i>Cumulative counter (type 2)</i>. The minimum, maximum, and average values are the same and represent the total accumulated value since the server was started. On server restart, counter values are reset to zero.
Units	Units of measure (bytes, percent, seconds, pieces, etc.)
Description	Counter description.

CPU counters (counters_vz_cpu class)

Counter ID	Value	Type	Units	Description
counter_cpu_system	int	2	seconds	System CPU time.
counter_cpu_user	int	2	seconds	User CPU time.
counter_cpu_idle	int	2	seconds	Idle CPU time.
counter_cpu_nice	int	2	seconds	Nice CPU time.
counter_cpu_starvation	int	2	seconds	'Starvation' CPU time (i.e. the difference between the guaranteed and used CPU time).
counter_cpu_system_states	int	0	percent	System CPU time in percent.
counter_cpu_user_states	int	0	percent	User CPU time in percent.
counter_cpu_idle_states	int	0	percent	Idle CPU time in percent.
counter_cpu_nice_states	int	0	percent	Nice CPU time in percent.
counter_cpu_starvation_states	int	0	percent	Starvation CPU time in percent.
counter_cpu_used	float	0	percent	Total CPU usage in percent.

counter_cpu_share_used	float	0	percent	The real CPU usage of the Container against the CPU limit set for this Container.
counter_cpu_limit	float	0	percent	The share of the CPU time the Container may never exceed.

UBC counters (counters_vz_abc class)

Counter ID	Value	Type	Units	Description
numproc	int	0	pcs	Number of processes and kernel-level threads.
numtcpsock	int	0	pcs	Number of TCP sockets.
numothersock	int	0	pcs	Number of non-TCP sockets.
vmguarpages	int	0	4K-pages	Memory allocation guarantee.
kmemsize	int	0	bytes	Size of non-swappable kernel memory.
tcpsndbuf	int	0	bytes	Total size of 'send' buffers for TCP sockets.
tcprecvbuf	int	0	bytes	Total size of 'receive' buffers for TCP sockets
othersockbuf	int	0	bytes	Total size of UNIX-domain socket buffers, UDP, and other datagram protocols 'send' buffers.
dgramrecvbuf	int	0	bytes	Total size of 'receive' buffers of UDP and other datagram protocols.
oomguarpages	int	0	4K-pages	Out-of-memory guarantee.
privvmpages	int	0	4K-pages	Size of the Container private memory.
lockedpages	int	0	4K-pages	Memory not allowed to be swapped out.
shmpages	int	0	4K-pages	Size of the shared memory.

physpages	int	0	4K-pages	Total size of RAM used by Container processes.
numfile	int	0	pcs	Number of open files.
numflock	int	0	pcs	Number of file locks.
numpty	int	0	pcs	Number of pseudo-terminals.
numsigninfo	int	0	pcs	Number of 'signinfo' structures.
dcachesize	int	0	bytes	Total size of 'dentry' and 'inode' structures locked in memory.
numiptent	int	0	pcs	Number of IP packet filtering entries.

Network counters (counters_vz_net class)

Counter ID	Value	Type	Units	Description
counter_net_incoming_bytes	int	2	bytes	The amount of incoming network traffic in bytes.
counter_net_incoming_packets	int	2	pcs	The amount of incoming network traffic in packets.
counter_net_outgoing_bytes	int	2	bytes	The amount of outgoing network traffic in bytes.
counter_net_outgoing_packets	int	2	pcs	The amount of outgoing network traffic in packets.

Disk quota counters (counters_vz_quota class)

Counter ID	Value	Type	Units	Description
diskspace	int	0	1K-blocks	The total size of disk consumed by the Container.
diskspace_hard	int	0	1K-blocks	Disk space hard limit.
diskspace_soft	int	0	1K-blocks	Disk space soft limit.
diskinodes	int	0	inodes	The total number of disk inodes (files, directories, symbolic links).
diskinodes_soft	int	0	inodes	The total number of disk inodes (files, directories, symbolic links). The Container is allowed to temporarily exceed the soft limit during the grace period defined by the 'quotatime' parameter.

diskinodes_hard	int	0	inodes	The total number of disk inodes (files, directories, symbolic links). The Container can never exceed this limit.
quotauidlimit	int	0	pcs	The number of user/group IDs allowed for Container internal disk quota. If set to 0, the UID/GID quota will not be enabled. You can configure the UID/GID quota for Containers with the disabled UID/GID quota only if they are stopped.
quotauidlimit_hard	int	0	pcs	The maximal number of user/group IDs allowed for Container internal disk quota.
counter_disk_used	int	0	bytes	The amount of disk space in use (in bytes).
counter_disk_share_used	float	0	percent	The ratio of the real disk space consumption by the Container against the disk space limit set for this Container.
counter_disk_limit	int	0	bytes	The total amount of disk space that can be consumed by the Container.

Load average counters (counters_vz_loadavg class)

Counter ID	Value	Type	Units	Description
counter_loadavg_l1	float	0	pcs	The average number of processes in the kernel run queue for the last minute.
counter_loadavg_l2	float	0	pcs	The average number of processes in the kernel run queue for the last 5 minutes.
counter_loadavg_l3	float	0	pcs	The average number of processes in the kernel run queue for the last 15 minutes.

System info counters (counters_vz_system class)

Counter ID	Value	Type	Units	Description
counter_system_users	int	0	number	Number of users.

counter_system_uptime	int	1	seconds	The time elapsed since the last server startup.
-----------------------	-----	---	---------	---

SLM counters (counters_vz_slm class)

Counter ID	Value	Type	Units	Description
slmmemorylimit	int	0	bytes	The total amount of memory that can be consumed by the Container.

Memory counters (counters_vz_memory class)

Counter ID	Value	Type	Units	Description
counter_memory_used	int	0	bytes	The total amount of memory used by the Container.
counter_memory_share_used	float	0	percent	The ratio of the real physical memory usage of the Container against the memory limit set for this Container, in percent.
counter_memory_limit	int	0	bytes	The total amount of memory that can be allocated to the Container.

Network counters (counters_vz_hw_net class)

Counter ID	Value	Type	Units	Description
counter_net_incoming_bytes	int	2	bytes	The amount of incoming network traffic in bytes.
counter_net_incoming_packets	int	2	pcs	The amount of incoming network traffic in packets.
counter_net_outgoing_bytes	int	2	bytes	The amount of outgoing network traffic in bytes.
counter_net_outgoing_packets	int	2	pcs	The amount of outgoing network traffic in packets.

Hardware Node CPU counters (counters_cpu class)

Counter ID	Value	Type	Units	Description
counter_cpu_system	int	2	seconds	System CPU time.
counter_cpu_user	int	2	seconds	User CPU time.
counter_cpu_nice	int	2	seconds	Nice CPU time.
counter_cpu_idle	int	2	seconds	Idle CPU time.
counter_cpu_system_states	int	0	percent	System CPU time in percent.
counter_cpu_user_states	int	0	percent	User CPU time percent.

counter_cpu_nice_states	int	0	percent	Nice CPU time in percent.
counter_cpu_idle_states	int	0	percent	Idle CPU time in percent.
counter_cpu_used	float	0	percent	CPU usage in percent.
counter_cpu_share_used	float	0	percent	The ratio of CPU time consumed by the server to current limit.
counter_cpu_limit	float	0	percent	CPU limit of the share the server will get.

Hardware Node disk counters (counters_disk class)

Counter ID	Value	Type	Units	Description
counter_disk_space_used	int	0	1K-blocks	Disk space used.
counter_disk_space_free	int	0	1K-blocks	Disk space free.
counter_disk_inodes_used	int	0	inodes	Disk inodes used.
counter_disk_inodes_free	int	0	inodes	Disk inodes free.
counter_disk_used	int	0	bytes	Disk space used in bytes.
counter_disk_share_used	float	0	percent	The ratio of used disk space to current limit.
counter_disk_limit	int	0	bytes	Total disk space available for the server.

Hardware Node memory counters (counters_memory class)

Counter ID	Value	Type	Units	Description
counter_memory_mem_used	int	0	bytes	Amount of used memory.
counter_memory_mem_free	int	0	bytes	Amount of available free memory.
counter_memory_swap_used	int	0	bytes	Amount of used swap.
counter_memory_swap_free	int	0	bytes	Amount of available free swap space.
counter_memory_used	int	0	bytes	Memory used by the server.
counter_memory_share_used	float	0	percent	The ratio of used memory to current limit.
counter_memory_limit	int	0	bytes	Total memory available for the server.

Hardware Node network counters (counters_net class)

Counter ID	Value	Type	Units	Description
counter_net_incoming_bytes	int	2	bytes	Amount of incoming network traffic in bytes.

counter_net_incoming_packets	int	2	pcs	Amount of incoming network traffic in packets.
counter_net_outgoing_bytes	int	2	bytes	Amount of outgoing network traffic in bytes.
counter_net_outgoing_packets	int	2	pcs	Amount of outgoing network traffic in packets.

Hardware Node load average counters (counters_loadavg class)

Counter ID	Value	Type	Units	Description
counter_loadavg_11	float	0	pcs	Average number of processes in the system run queue of kernel for the last 1 minute.
counter_loadavg_12	float	0	pcs	Average number of processes in the system run queue of kernel for the last 5 minutes.
counter_loadavg_13	float	0	pcs	Average number of processes in the system run queue of kernel for the last 15 minutes.

Hardware Node system info counters (counters_system class)

Counter ID	Value	Type	Units	Description
counter_system_uptime	int	1	seconds	Processor uptime.
counter_system_users	int	0	number	Number of users.

Index

A

About This Guide • 4
 Advanced Topics • 142
 Agent Architecture • 13
 Agent Configuration • 142
 Agent Messages • 19
 Apache Axis 1.2 for Java • 140
 Appendix A
 Performance Counters • 147
 Authentication Concepts • 15
 Authorization • 16

B

Backing up a Container • 117
 Backup Operations • 117
 Base64-encoded Values • 95

C

Certificates Policy Preparation • 81
 Classes, Instances, Counters • 67, 132
 Cloning a Virtuozzo Container • 112
 Complete Program Code • 88
 Configuring a Container • 106
 Configuring a Virtuozzo Container • 63
 Connecting to Agent • 33
 Connection URL • 82
 Connectivity • 14
 Creating a Container • 99
 Creating a Simple Client Application • 32, 79
 Creating a Virtuozzo Container • 60
 Creating and Configuring Virtuozzo Containers • 54

D

Destroying a Container • 103
 Destroying a Virtuozzo Container • 66
 Developing Agent SOAP Clients • 93
 Documentation Conventions • 5
 Dynamic Limits • 145

E

Elements with no Content • 95
 Error Handling • 31
 Events and Alerts • 72

F

Feedback • 7

G

General Conventions • 7
 Generating Client Code from WSDL • 79
 Get/Set Method Name Conflict • 97
 Getting a List of OS Templates • 57
 Getting a List of Sample Configurations • 55
 Getting a Performance Report • 68, 133
 Getting Container Configuration Information • 105
 Getting Container Information From a Backup • 130
 Getting Started • 8

I

Installation • 10
 Internal Request Scheduler • 142
 Introduction • 78

K

Key Features • 79

L

Limitations • 79
 Listing Backups • 120
 Location of XSD and WSDL • 12
 Logging In • 35, 47
 Logging in and Creating a Session • 84
 Logging In To VZCC or VZPP • 51
 Login and Session Management • 43

M

Managing Virtuozzo Containers • 98
 Message Body • 27
 Message Classification and Priorities • 143
 Message Header • 22
 Migrating a Container to a Different Host • 114
 Modifying Container Name • 109
 Modifying DNS Server Assignment • 111
 Modifying Hostname • 108
 Modifying IP Address • 106
 Modifying QoS Settings • 110

Monitoring Alerts • 136
Monitoring Multiple Environments • 71
Multiple Calls and Targets • 29

O

Optional Elements • 94
Organization of This Guide • 5
Other SOAP Clients and Their Known Issues •
139
Overview • 78

P

Parallels Agent API • 9
Parallels Agent Overview • 8
Passing parameters explicitly • 64
Performance Monitor • 66, 132
Pool and Single Operators • 144
Populating Container Configuration Structure •
58
Preface • 4

Q

Queue • 145

R

Realms • 15
Receiving Periodic Reports • 70
Request Routing • 75
Restarting a Virtuozzo Container • 39
Restoring a Container • 126
Retrieving a List of Virtuozzo Containers • 38,
86
Retrieving Container Configuration • 62
Retrieving Realm Information • 44

S

Sessions • 49
Shell Prompts in Command Examples • 7
SOAP API Reference • 93
SOAP Object Binding • 83
Starting, Stopping, Restarting • 11
Starting, Stopping, Restarting a Container •
102
Step 1
 Choosing a Development Project • 80
Step 2
 Generating Proxy Classes From WSDL • 80
Step 3
 Main Program File • 81
Step 4
 Running the Sample • 87
Summary • 40
Suspending and Resuming a Container • 104
System Requirements • 10

T

Terminology • 16
The Complete Program Code • 41
The Null-Terminating Character • 30
Timeouts • 96, 146
Troubleshooting • 141
Typographical Conventions • 6

U

Using SOAP API • 78
Using values from a sample configuration • 65
Using XML API • 18

V

Visual Basic .NET • 139
Visual J# .NET • 139

W

Who Should Read This Guide • 4

X

XML API Basics • 18
XML Message Examples • 21
XML Message Specifications • 19
XML Schema • 19