HP Diagnostics

Software Version: 9.24

.NET Agent Guide

Document Release Date: January 2015
Software Release Date: January 2015

Legal Notices

Warranty

The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

The information contained herein is subject to change without notice.

Restricted Rights Legend

Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Copyright Notice

© Copyright 2005 - 2015 Hewlett-Packard Development Company, L.P.

Trademark Notices

Adobe™ is a trademark of Adobe Systems Incorporated.

Microsoft® and Windows® are U.S. registered trademarks of Microsoft Corporation.

UNIX® is a registered trademark of The Open Group.

Java is a registered trademark of Oracle and/or its affiliates.

Oracle® is a registered trademark of Oracle and/or its affiliates.

Acknowledgements

This product includes software developed by the Apache Software Foundation (http://www.apache.org/).

This product includes software developed by the Spice Group (http://spice.codehaus.org).

For information about open source and third-party license agreements, see the *Open Source and Third-Party Software License Agreements* document in the Documentation directory on the product installation media.

Documentation Updates

The title page of this document contains the following identifying information:

- Software Version number, which indicates the software version.
- Document Release Date, which changes each time the document is updated.
- Software Release Date, which indicates the release date of this version of the software.

To check for recent updates or to verify that you are using the most recent edition of a document, go to: https://softwaresupport.hp.com/group/softwaresupport/search-result?keyword=.

This site requires an HP Passport account. If you do not have one, click the **Create an account** button on the HP Passport Sign in page.

Support

Visit the HP Software Support web site at: **https://softwaresupport.hp.com**

This web site provides contact information and details about the products, services, and support that HP Software offers.

HP Software Support provides customer self-solve capabilities. It provides a fast and efficient way to access interactive technical support tools needed to manage your business. As a valued support customer, you can benefit by using the support web site to:

- Search for knowledge documents of interest
- Submit and track support cases and enhancement requests
- Download software patches
- Manage support contracts
- Look up HP support contacts

- Review information about available services
- Enter into discussions with other software customers
- Research and register for software training

Most of the support areas require that you register as an HP Passport user and sign in. Many also require a support contract. To register for an HP Passport ID, go to **https://softwaresupport.hp.com** and click **Register**.

To find more information about access levels, go to: **https://softwaresupport.hp.com/web/softwaresupport/access-levels**

HP Software Solutions & Integrations and Best Practices

Visit HP Software Solutions Now at **https://h20230.www2.hp.com/sc/solutions/index.jsp** to explore how the products in the HP Software catalog work together, exchange information, and solve business needs.

Visit the Cross Portfolio Best Practices Library at **https://hpln.hp.com/group/best-practices-hpsw** to access a wide variety of best practice documents and materials.

Contents

Welcome to This Guide 9

How This Guide Is Organized 9

Diagnostics Documentation 10

Part 1: Introduction 11

Chapter 1: Diagnostics .NET Agent Overview 12

About the Diagnostics .NET Agent 12

Introducing Diagnostics Profiler for .NET 12

Features and Benefits of the Diagnostics .NET Profiler 13

Part 2: Installation and Configuration of the Diagnostics .NET Agent 14

Chapter 2: Preparing to Install the Diagnostics .NET Agent 15

Requirements for the Diagnostics .NET Agent Host 15

Requirements for the Diagnostics .NET Profiler UI 16

Planning the Installation 16

Chapter 3: Installing .NET Agents 18

Overview of the .NET Agent Installation 19

Accessing the .NET Agent Installer 20

Installing the .NET Agent 21

Post Install Tasks 38

Verifying the .NET Agent Installation 38

About Configuration of the .NET Agent for Diagnostics 39

About Configuration of the .NET Agent for TransactionVision 39

Discovery and Standard Instrumentation 41

Probe Aggregator Service 44

Monitoring NET Applications Deployed in Azure Cloud 45

Monitoring Applications on SharePoint with the .NET Agent 45

Determining the Version of the .NET Agent 47

Enabling and Disabling the Diagnostics Agent for .NET 48

Enabling and Disabling Standard Instrumentation for Applications 48

Troubleshooting .NET Web Applications Not Discovered 50

Manually Adding an AppDomain Not Discovered 51

Other .NET Agent Troubleshooting Tips 54

Uninstalling the .NET Agent 55

Chapter 4: Upgrading the Diagnostics .NET Agent 56
Upgrade .NET Agents 56
Upgrade Notes and Limitations 57

Part 3: Advanced .NET Agent Configuration and Instrumentation 58

Chapter 5: Custom Instrumentation for .NET Applications 59
About Instrumentation and Capture Points Files 59
Locating the .NET Capture Points Files 60
Coding Points in the Capture Points File 61
Instrumentation Examples 66
Understanding the Overhead of Custom Instrumentation 83
Default Layers for Typical .NET Applications 83

Chapter 6: Understanding the .NET Agent Configuration File 85
Understanding .NET Agent Configuration File 85
.NET Agent Configuration Elements 86
<ali> element 86
<appdomain> element 87
<bufferpool> element 89
<capturecookies> element 90
<captureexceptions> element 91
<capturehttpheaders> element 92
<clientmonitoring> element 93
<consumeridrules> element 94
<cputime> element 95
<credentials> element 96
<demomode> element 97
<depth> element 98
<diagnosticsserver> element 99
<exceptiontype> element 101
<exclude> element (when parent is captureexceptions) 102
<exclude> element (when parent is lwmd) 103
<excludeassembly> element 104
<excludesqlparam> element 105
<filter> element 106
<filter> element 107
<gentvhttpeventforwcf> element 108
<htmlinstrumentation> element 109
<httpcaptureparams> element 110
<httpclient> element 112

<httpheaderrule> element 112
<httpheaderrules> element 114
<id> element 115
<include> element (when parent is captureexceptions) 116
<include> element (when parent is lwmd) 117
<instrumentation> element 118
<iprule> element 119
<iprules> element 120
<latency> element 121
<logging> element (when parent is appdomain, probeconfig, or process) 125
<lwmd> element 127
<mediator> element 128
<metrics> element 130
<metric> element 131
<modes> element 133
<param> element 136
<points> element 137
<probeconfig> element 138
<process> element 139
<profiler> element 141
<rum> element 143
<sample> element 145
<server> element 146
<soapcapture> element 147
<soaprequestforsoapfault> element 148
<soaprule> element 149
<soaprules> element 150
<sqlparsing> element 151
<stacktracesampling> element 152
<symbols> element 154
<timeskew> element 156
<topology> element 157
<transport> element 158
<trim> element 160
<tv> element 161
<uriautocollapsing> element 162
<urireplacepattern> element 164
<url> element 165
<vmware> element 166
<webserver> element 167

<ws> element 168
<xvm> element 169

Chapter 7: Advanced .NET Agent Configuration 170
Time Synchronization for .NET Agents Running on VMware 171
Customizing the Instrumentation for ASP.NET Applications 171
Discovering the Classes and Methods in an Application 175
Controlling Which HP Software Products the Agent can Work With 177
Configuring Support for MSMQ BasedCommunication 179
Configuring Latency Trimming and Throttling 179
Configuring Depth Trimming 182
Configuring URI Truncation and Mapping 183
Capturing HTTP Server Requests Based on Query Parameters 184
Configuring the .NET Agent for Lightweight Memory Diagnostics 186
Limiting Exception Stack Trace Data 188
Configuring Thread Stack Trace Sampling 190
Disabling Logging 192
Overriding the Default Probe Host Machine Name 192
Listing the Probes Running on a Host 193
Authentication and Authorization for .NET Profilers 194
Configuring Consumer IDs 195
Configuring SOAP Fault Data 199
Collecting Additional Probe Metrics or Modifying Probe Metrics 199
Manually Enabling Auto-Discovered ASP.NET Applications and Non ASP.NET Services 201

Chapter 8: .NET System Metrics Agent - Systems Metrics Capture 202
About the .NET System Metrics Agent 202
System Metrics Captured by Default 202
Configuring .NET System Metrics Capture 203
Adding System Metrics Using the Windows Performance Monitor 205
Default Entries in the .NET Agent metrics.config File 207
Keywords in the metrics.config File 208

Part 4: Using the Profiler for .NET 211

Chapter 9: Diagnostics Profiler for .NET 212
About the .NET Diagnostics Profiler 212
How the .NET Agent Provides Data for the .NET Profiler 213
.NET Diagnostics Profiler UI Navigation and Display Controls 214
.NET Diagnostics Profiler Inactivity Timeout 215
How to Access the .NET Diagnostics Profiler 215
How to Enable and Disable the .NET Diagnostics Profiler 216

Server Requests Tab Description 217
SQL Tab Description 220
Methods Tab Description 223
Call Tree Tab Description 225
Exceptions Tab Description 228
Collections Tab Description 230
Threads Window Description 233

Send Documentation Feedback 239

Welcome to This Guide

Welcome to the HP Diagnostics .NET Agent Guide. This guide describes how to install, configure and use the Diagnostics .NET Agent and the Diagnostics Profiler for .NET.

The Diagnostics .NET Agent captures events such as method invocations, collection sites, and the beginning and end of business and server transactions.

The .NET Agent works with many of HP Software's Diagnostics products such as LoadRunner, Business Availability Center, and Performance Center and is an integrated part of HP Software's application lifecycle solution which includes load testing, production monitoring, and trouble diagnosis.

The Diagnostics Profiler for .NET is installed as part of the Diagnostics .NET Agent. The Diagnostics Profiler for .NET provides a way for .NET development teams to monitor the performance and diagnose issues with applications in the development environment. HP Software makes this tool available at no cost, through an easy-to-install trial software download.

How This Guide Is Organized

This guide contains the following parts:

- **"Introduction" on page 11**

  Provides a high level overview of the features, components, architecture, and outputs of the Diagnostics .NET Agent and Diagnostics Profiler for .NET.

- **"Installation and Configuration of the Diagnostics .NET Agent" on page 14**

  Describes how to install and configure the Diagnostics Agent.

- **"Advanced .NET Agent Configuration and Instrumentation" on page 58**

  Describes advanced configuration of the .NET Agent.

- **"Using the Profiler for .NET" on page 211**

  Describes the UI of the Diagnostics .NET Profiler, and how to use it.

Diagnostics Documentation

HP Diagnostics includes the following documentation. Unless specified otherwise, the guides are in PDF format only and are available as downloads from the HP Software Support site (at https://softwaresupport.hp.com).

Diagnostics User Guide and Online Help. Explains how to choose and interpret the Diagnostics views in the Diagnostics Enterprise UI to analyze your monitored applications. To access the online help for Diagnostics, choose **Help** > **Help** in the Diagnostics Enterprise UI. If Diagnostics is integrated with another HP Software product the online help is also available through that product's Help menu. The User Guide is a PDF version of the online help and their content is identical. The User Guide is available from the Diagnostics online help Home page, from the Windows Start menu (**Start > Programs > HP Diagnostics Server > User Guide)**, or from the Diagnostics Server installation directory.

Diagnostics Server Installation and Administration Guide. Explains how to plan a Diagnostics deployment, and how to install and maintain a Diagnostics Server.

The following Agent guides contain content that supports agent installation, setup and configuration.

- **Diagnostics Java Agent Guide.** Describes how to install, configure, and use the Diagnostics Java Agent and the Diagnostics Profiler for Java.
- **Diagnostics .NET Agent Guide.** Describes how to install, configure, and use the Diagnostics .NET Agent and Diagnostics Profiler for .NET.
- **Diagnostics Python Agent Guide.** Describes how to install, configure, and use the Diagnostics Python Agent. The Guide (in PDF format) is also available from the Profiler UI help.

Diagnostics Collector Guide. Explains how to install and configure a Diagnostics Collector.

Diagnostics System Requirements and Support Matrixes Guide. Describes the system requirements for the various Diagnostics components.

Release Notes. Provides last-minute new information and known issues about each version of Diagnostics. The PDF file is also located in the Diagnostics installation disk root directory.

Diagnostics Data Model and Query API. Describes the Diagnostics data model and the query API you can use to access the data. The guide is also available from the Diagnostics online help Home page.

Diagnostics Frequently Asked Questions (FAQ). Gives answers to frequently asked questions. The FAQ is also available from the Diagnostics online help Home page.

Part 1: Introduction

Chapter 1: Diagnostics .NET Agent Overview

This chapter introduces the Diagnostics .NET Agent and the Diagnostics Profiler for .NET by providing a high level overview of features and components.

This chapter includes:

- "About the Diagnostics .NET Agent" below
- "Introducing Diagnostics Profiler for .NET " below
- "Features and Benefits of the Diagnostics .NET Profiler" on the next page

About the Diagnostics .NET Agent

The Diagnostics .NET Agent is installed on the machine that hosts the application that you want to monitor. Agent installation and setup automatically discovers and provides standard instrumentation for the .NET AppDomains you choose to monitor.

The agent captures events such as method invocations, collection sites, and the beginning and end of business and server transactions.

The .NET Agent works with many of HP Software's Diagnostics products such as LoadRunner, Performance Center and BSM.

Introducing Diagnostics Profiler for .NET

The Diagnostics Profiler for .NET is installed as part of the Diagnostics .NET Agent.

The Diagnostics Profiler for .NET provides a way for .NET development teams to monitor the performance and diagnose issues with applications in the development environment. HP Software makes this tool available at no cost, through an easy-to-install download.

The Diagnostics Profiler for .NET provides a strong foundation for collaborative diagnostics because it has been built using the same Diagnostics probe technology that is used in HP Software's load testing and production monitoring products. When you use the Diagnostics .NET Profiler in the development environment to profile applications and solve problems, you get a glimpse of the features that are included in the Diagnostics Lifecycle Solution that enable you to solve the toughest performance problems throughout the application's lifecycle.

Because Diagnostics Profiler for .NET uses the same agent that other HP Software Diagnostics products use, it is an integrated part of HP Software's application lifecycle solution which includes load testing, production monitoring, and trouble diagnosis.

Features and Benefits of the Diagnostics .NET Profiler

The following table describes some of the features and benefits of the Diagnostics .NET Agent and the Diagnostics Profiler for .NET:

Feature Description	Benefit
Server Request Breakdown	Identify where time is spent in an application
Layer Breakdown	Identify the slowest layer
Slowest Server Requests	Identify slowest server request or application entry points
Top 3 Slowest Instances	Identify outliers to help diagnose intermittent problems
VM Heap Usage	Identify memory problems and garbage collection issues
Collection Memory Leak Diagnostics	Identify the fastest growing and largest size collections including the caller method that allocated the collection
Heap Breakdown including Class and Size Information	Identify leaking objects, object growth trends, object instance counts, and the byte size for objects
SQL Diagnostics (Slowest SQL)	Identify the slowest SQL query and report query information
Exception Diagnostics	Identify exception counts which often go undetected
Snapshot	Capture all the data displayed on all the tabs into a single XML report that can be stored or transported for later viewing and analysis.

Part 2: Installation and Configuration of the Diagnostics .NET Agent

Chapter 2: Preparing to Install the Diagnostics .NET Agent

This chapter provides you with the information and instructions that will help you to plan and prepare for the installation and configuration of the Diagnostics .NET Agent.

If you are installing the agent for use in an HP AppPulse environment please refer to the *HP Diagnostics .NET Agent Quick Start Guide* (Diagnostics_Dotnet_Agent_QuickStart.pdf) for installation instructions. This document is provided in the HP AppPulse UI for download with the agent software.

This chapter includes:

- "Requirements for the Diagnostics .NET Agent Host" below
- "Requirements for the Diagnostics .NET Profiler UI" on the next page
- "Planning the Installation" on the next page

Requirements for the Diagnostics .NET Agent Host

For details of the Diagnostics .NET Agent host requirements, see "Requirements for the Diagnostics .NET Agent Host" in the Diagnostics System Requirements and Support Matrixes Guide located on the HP Software Support site. Access requires an HP Passport login (register for an HP Passport).

Requirements for the Diagnostics .NET Profiler UI

For details of the Diagnostics .NET Profiler UI requirements, see "Requirements for the Diagnostics .NET Profiler UI" in the Diagnostics System Requirements and Support Matrixes Guide located on the HP Software Support site. Access requires an HP Passport login (register for an HP Passport).

Planning the Installation

The .NET Agent is installed on the same machine as the .NET application under test. The following table is provided to help you gather the information that you will need during the installation of the .NET Agent.

Diagnostics Server Information

Information Required	Where to find it	Value
Mode for installing the agent	Choose according to product license.	• Profiler only (no connection to server) • Used only with LoadRunner/Performance Center (AD license) • Enterprise mode (AM license) for use with one of the following or both: • Diagnostics • TransactionVision
Diagnostics Server Name	Fully qualified host name or IP address of the host of the Diagnostics Server. System Health Monitor. (See "Using System Views for Administrators" in the HP Diagnostics Server Installation and Administration Guide.) This is not required for using the .NET Diagnostics Profiler in a standalone mode.	If there is only one Diagnostics Server in the deployment where the agent will run, this is the Diagnostics Server in Commander mode. In a distributed environment with a commander server and mediator servers, this is the Diagnostics Server in Mediator mode that is to receive the events from the agent.

Information Required	Where to find it	Value
Diagnostics Server Port	System Health Monitor. This is not required for using the.NET Diagnostics Profiler in a standalone mode.	**Default value:** `2612`

Agent and Port Information

Information Required	Where to find it	Value
agent group	This is user defined at the time that the agent is installed. The agent group name you enter is used as the probe group name Probe groups are logical groupings of probes that report to the same Diagnostics Server.	Default value: `Default`
Web Port Min	System Administrator. The lowest port number in a range of ports on the agent system that can be assigned to the probe.	Default value: `35000`
Web Port Max	System Administrator. The highest port number in a range of ports on the agent system that can be assigned to the probe.	Default value: `35100`

Chapter 3: Installing .NET Agents

This section describes how to install a .NET Agent and gives you information about the setup and configuration of the .NET Agent.

If you are installing the agent for use in an HP AppPulse environment please refer to the *HP Diagnostics .NET Agent Quick Start Guide* (Diagnostics_Dotnet_Agent_QuickStart.pdf) for installation instructions. This document is provided in the HP AppPulse UI for download with the agent software.

This chapter includes:

- "Overview of the .NET Agent Installation" on the next page
- "Accessing the .NET Agent Installer" on page 20
- "Installing the .NET Agent" on page 21
- "Post Install Tasks" on page 38
- "Verifying the .NET Agent Installation" on page 38
- "About Configuration of the .NET Agent for Diagnostics" on page 39
- "About Configuration of the .NET Agent for TransactionVision" on page 39
- "Discovery and Standard Instrumentation" on page 41
- "Probe Aggregator Service" on page 44
- "Monitoring NET Applications Deployed in Azure Cloud" on page 45
- "Monitoring Applications on SharePoint with the .NET Agent" on page 45
- "Determining the Version of the .NET Agent" on page 47
- "Enabling and Disabling the Diagnostics Agent for .NET" on page 48
- "Enabling and Disabling Standard Instrumentation for Applications" on page 48
- "Troubleshooting .NET Web Applications Not Discovered" on page 50
- "Manually Adding an AppDomain Not Discovered" on page 51
- "Other .NET Agent Troubleshooting Tips" on page 54
- "Uninstalling the .NET Agent" on page 55

Overview of the .NET Agent Installation

The .NET Agent software is installed on the machine hosting the application you want to monitor. With the .NET Agent you instrument the application domains for monitoring.

See "Preparing to Install the Diagnostics .NET Agent" on page 15 for .NET Agent requirements.

The .NET Agent (version 9.x) requires .NET Framework 2.0 or later. The .NET Framework must be installed on the machine before you run the .NET Agent installation.

> **Note:** If you need to support .NET Framework 1.1, you will need to use an earlier version of the .NET Agent (8.x).

WCF Requirements and Limitations: Monitoring .NET Windows Communication Foundation (WCF) services requires .NET Framework 3.0 SP1 or greater. WCF bindings using the following transports are supported:

- HTTP
- TCP

If your application uses a transport that is not supported, the .NET probe only creates a generic server request for each WCF method. It will not be a Web Service and there will be no cross VM correlation.

The HP Diagnostics/TransactionVision .NET Agent installer installs a .NET Agent to collect data for either Diagnostics or TransactionVision or both.

The .NET Agent installer automatically detects the ASP.NET applications on the system where the agent is installed. See "Discovery and Standard Instrumentation" on page 41.

The installer configures the agent to capture basic workload and events for each of the ASP.NET applications detected. The agent configuration is controlled using the **probe_config.xml** file. See "Automatic Instrumentation and Configuration for Discovered ASP.NET Applications" on page 42.

The .NET agent uses **points files** to provide standard instrumentation to enable you to start monitoring applications. The points files control the workload the agent captures for the application. See "Custom Instrumentation for .NET Applications" on page 59. See "Enabling and Disabling Standard Instrumentation for Applications" on page 48.

The following points files are installed and enabled to provide instrumentation for monitoring ASP.NET applications:

- ASP.NET.points
- ADO.points
- WCF.points

The following points files can be used for instrumenting applications that use other Microsoft technologies:

- Remoting.points (for .NET remoting environments)
- msmq.points (for MSMQ environments)
- LWMD.points (for analysis of memory used by collections in applications)

Separate instrumentation points files are created for each IIS installed ASP.NET application domain detected and named **<AppDomain>.points** files). The **probe_config.xml** file contains an <appdomain> element for each of the detected ASP.NET applications. And each <appdomain> element contains an instrumentation points file reference. The .NET Agent uses this runtime instrumentation to capture method latency information from specified applications.

Note: If there is a pre-existing installation of the .NET Agent on the host machine see "Upgrade .NET Agents" on page 56 for important instructions on how to upgrade the agent systems.

See "Accessing the .NET Agent Installer" below to begin.

Accessing the .NET Agent Installer

You can launch the .NET Agent installer a number of different ways. You can install the .NET Agent from the Diagnostics installation disk or the BSM installation disk or from the Downloads page in BSM. You can install the software from the SSO Portal. And if you want to install a trial version of the HP Diagnostics Profiler for .NET you can launch the installer from the HP Software Web site download center.

Note: If you are installing the agent for use in an HP AppPulse environment please refer to the *HP Diagnostics .NET Agent Quick Start Guide* (Diagnostics_Dotnet_Agent_QuickStart.pdf) for installation instructions. This document is provided in the HP AppPulse UI for download with the agent software.

To access the Installer from a Diagnostics installation location:

- From the Diagnostics Installation DVD (Autorun.exe) the installation menu page is displayed. From the menu, select **Diagnostics Agent for .NET 64-bit** to launch the install for the 64-bit version of the .NET agent.
- You could run the appropriate installer directly by locating the executable file **HPDiagTV.NETAgt_<release number>_win64.msi** in the location you install from and copying the file to the new installation location and then double-clicking it to run the installer.

Continue with "Installing the .NET Agent" on the next page.

To download the installer from the HP Software Download Center:

1. Access the SSO portal at http://support.openview.hp.com/selfsolve using your HP Passport login.
2. Locate the Diagnostics (or TransactionVision) downloads and choose the appropriate link for downloading the Diagnostics .NET Agent software. Note that you could also use the download center in order to get the Diagnostics .NET profiler trial/evaluation software.
3. Continue with "Installing the .NET Agent" below.

   Follow the download instructions on the web site.

To download the Installer from BSM's Diagnostics downloads page:

1. In **BSM**, either select **Admin > Diagnostics** from the main menu and click the **Downloads** tab. Or select **Admin > Platform** from the main menu and click the **Setup and Maintenance** tab.
2. On the Downloads page, click the appropriate link to download the .NET Agent installer for 64-bit Windows.

Note: The .NET Agent installers are available in BSM if put into the required directory for BSM to access. You can enable this during the installation of the Diagnostic Server, or you can copy the .NET agent installers manually from the Diagnostics installation disk to the required location.

Continue with "Installing the .NET Agent" below.

To launch the installer for HP Diagnostics Profiler for .NET trial software from the HP Software Trial Software Download Web site:

1. Go to the HP Software Web site's Download Center.
2. In the **Quick Search** section, in the **Products** list, click **Diagnostics** and click **Search**.
3. Under **Trial Software**, select the appropriate link.
4. Follow the download instructions on the web site.

Continue with "Installing the .NET Agent" below.

Installing the .NET Agent

This section provides detailed instructions for a *first time* installation of the .NET Agent. If there is a pre-existing installation of the .NET Agent on the host machine see "Upgrading the Diagnostics .NET Agent" on page 56 for important instructions on how to upgrade the agent systems.

Caution: If the machine on which you are installing the .NET Agent already has a non

HP monitoring or profiling tool installed on it, the .NET Agent setup program detects this installation and provides the following options:

- To cancel the .NET Agent installation so that you can manually uninstall the other tool and then restart the .NET Agent installation.
- To continue with the .NET Agent installation. Note that having both the HP Diagnostics .NET Agent and a non HP monitoring or profiling tool installed on the same machine may result in the following:
  - Should you decide to uninstall the other (non HP) monitoring or profiling tool after installing the .NET Agent, this may adversely affect the .NET Agent and if so, requires running the **Enable .NET Agent** option from **Start** menu.
  - The other (non HP) monitoring or profiling tool may not function correctly.

Note: If you are installing the agent for use in an HP AppPulse environment please refer to the *HP Diagnostics .NET Agent Quick Start Guide* (Diagnostics_Dotnet_Agent_QuickStart.pdf) for installation instructions. This document is provided in the HP AppPulse UI for download with the agent software.

The .NET Agent installation process includes the following steps (select "Step 1. End user license agreement" below to begin):

"Step 1. End user license agreement" below

"Step 2. Specify install location" on the next page

"Step 3. Select installation options" on the next page

"Step 4. Specify RUM Integration Settings" on page 24

"Step 5. Select agent features to install" on page 26

"Step 6. Agent name and group" on page 26

"Step 7. Diagnostics server information" on page 29

"Step 8. Port and connection information" on page 30

"Step 9. Pre-install summary" on page 35

"Step 10. Additional Setup for Agents Working in an HP SaaS Environment" on page 35

"Step 11. Post Install Information" on page 37

"Step 12. Restart IIS" on page 37

Step 1. End user license agreement

Accept the end user license agreement.

Read the agreement and select **I accept the terms of the License Agreement**.

Click **Next** to proceed and continue to the next step.

Step 2. Specify install location

Provide the location where you want the Agent installed.

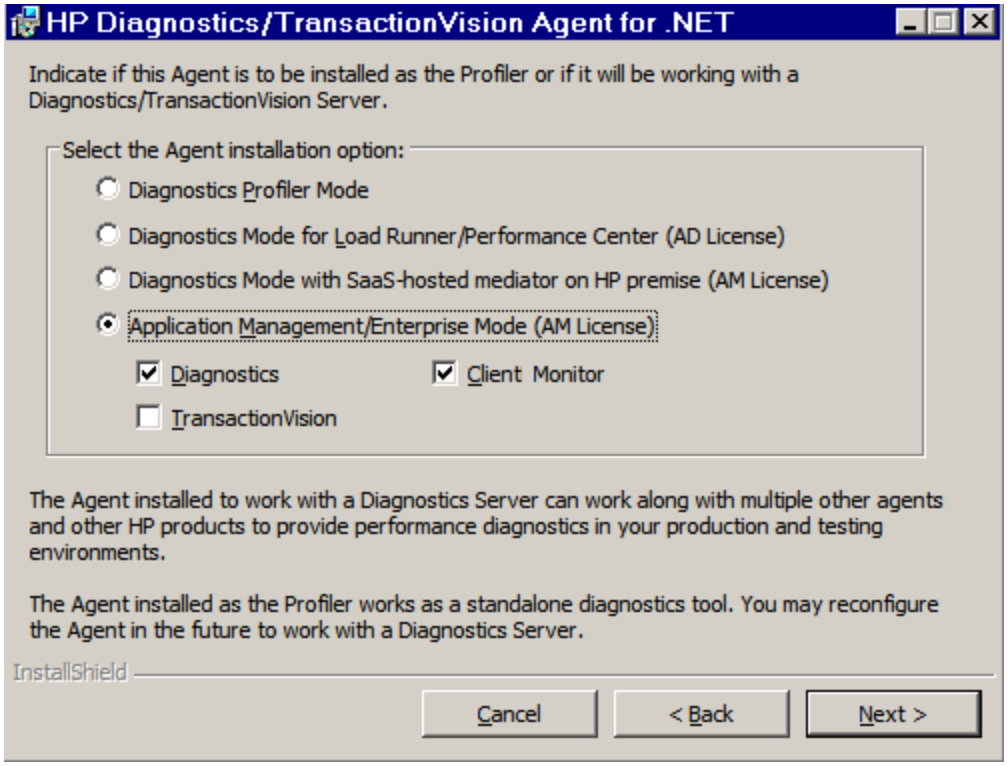
By default, the Agent is installed in **C:\MercuryDiagnostics\.NET Probe**. This location becomes the <probe_install_dir>.

Accept the default directory or select a different location either by typing in a different path, or by clicking **Browse** to navigate to the installation directory.

Click **Next** to proceed and continue to the next step.

Step 3. Select installation options

Indicate if the .NET Agent is to be installed as a standalone Profiler without any connection to a server (for example if you are installing the Diagnostics .NET Profiler trial software), or if you are installing the agent to work for LoadRunner/Performance Center or to work with a Diagnostics and/or TransactionVision Server and/or RUM Client Monitor.



Make the selection that is appropriate for the environment where you will be using the agent.

Diagnostics Profiler Mode. Select this option to install the agent as a Diagnostics .NET Profiler without any connection to a Diagnostics server. This is typically selected when installing the Diagnostics .NET Profiler trial software prior to purchasing the HP Diagnostics product.

If you select Diagnostics Profiler Mode option, the value of the **probe_config.xml <modes>** element is set to **pro** mode at the time you install the .NET Agent (see "<modes> element" on page 133).

Diagnostics Mode for LoadRunner/Performance Center (AD License). Select this option to install the agent for use with a Diagnostics Server in a load testing (or pre-production) environment where probes are used only in LoadRunner or Performance Center runs.

The advantage of running a probe in AD mode is that probes in AD mode are only counted against your HP Diagnostics AD license capacity when in a LoadRunner or Performance Center run. For example if you have 20 probes installed in LoadRunner/Performance Center AD mode but only 5 in a run, then only 5 are counted against your AD license capacity.

In AD mode the agent will ONLY capture data during a LoadRunner or Performance Center run and the results will be stored in a specific Diagnostics database for that run, for example, Default Client:21. When the agent is in AD mode it will not use resources or send any data to the server unless the probe is part of a LoadRunner/Performance Center run.

If you select this AD License option, the value of the **probe_config.xml<modes>** element is set to **ad** mode at the time you install the .NET Agent (see "<modes> element" on page 133).

See the chapter "Licensing HP Diagnostics" in the HP Diagnostics Server Installation and Administration Guide for more information.

Diagnostics Mode with SaaS-hosted mediator on HP premise (AM License). Select this option to install the agent to work in a SaaS environment where the .NET agent will connect to an HP SaaS server on-premise at HP. An HP SaaS administrator will provide you with information on connecting the .NET agent to an HP SaaS hosted Diagnostics mediator server.

Application Management/Enterprise Mode (AM License). Select this option to install the agent for use with a Diagnostics Server and/or a TransactionVision Server in an enterprise (or production) environment and/or RUM Client Monitor.

Then indicate which of the following the agent will be configured for:

- A Diagnostics Server (installed locally)
- A TransactionVision server

  See the *HP TransactionVision Deployment Guide* in the BSM documentation library for details on setup options specific to TransactionVision.

- RUM Client Monitor

  Enables the integration between Diagnostics and Real User Monitor (RUM).

With this option, the value of the **probe_config.xml<modes>** element is set to **enterprise** mode if you select the Diagnostics Server and **tv** mode if you select the TransactionVision server at the time you install the .NET Agent (see "<modes> element" on page 133).

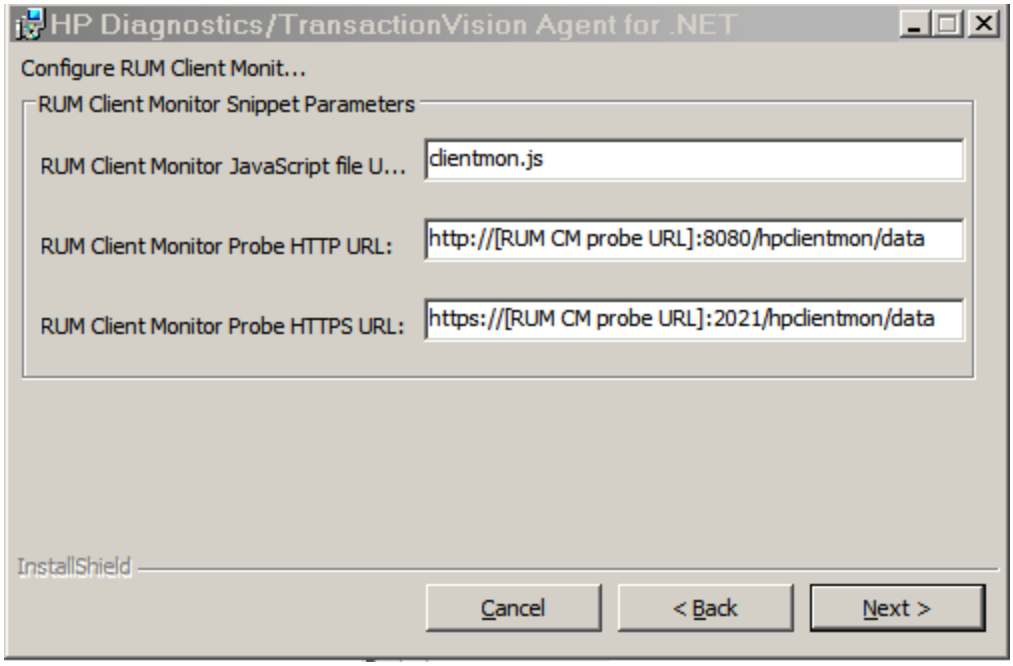
For those agents with Enterprise mode set, the agent will be counted against your HP Diagnostics AM license capacity.

Click **Next** to proceed and continue to the next step

Step 4. Specify RUM Integration Settings

This step is skipped if the RUM Client Monitor check box is not selected in "Step 3. Select installation options" on the previous page.

Enter the configuration information for the RUM Client Monitor JavaScript snippet.



- **RUM Client Monitor JavaScript file URL.** Enter the full URL path to the source file containing the RUM Client Monitor JavaScript. The default file name is clientmon.js.

  Note: Copy the RUM JavaScript (clientmon.js) from the RUM installation package. Save it on the .NET IIS Application Server in the root directory of the web application which is being monitored.

- **RUM Client Monitor Probe HTTP URL.** Enter the URL of the RUM Browser Probe to which the monitored client data is sent. The format for the URL is: <protocol>://<host>:<port>/hpclientmon/data

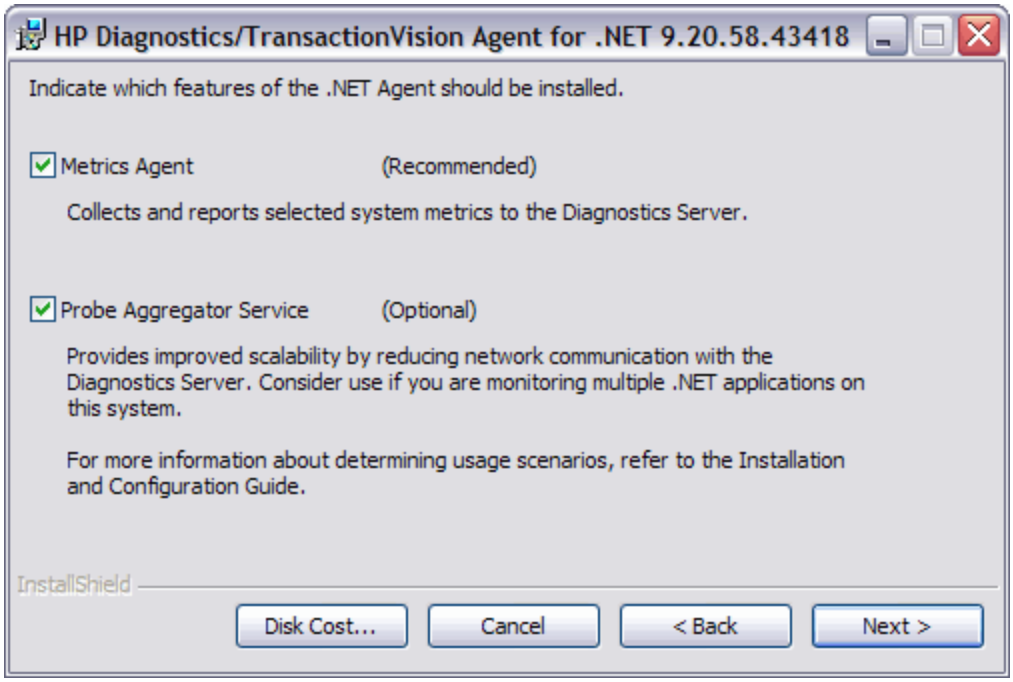
- **RUM Client Monitor Probe HTTPS URL.** Enter the URL of the RUM Browser Probe to which the monitored client data is sent, if using https. The format for the URL is: <protocol>://<host>:<port>/hpclientmon/data

Click **Next** to proceed and continue to the next step.

Note: For details on the RUM Client Monitor-Diagnostics integration, including how to configure these settings manually, refer to the RUM Client Monitor-Diagnostics Integration Guide located on the HP Software Support site. Access requires an HP Passport login (register for an HP Passport).

Step 5. Select agent features to install

Select the .NET Agent features you want to install.



Metrics Agent. It is recommended that you install the Metrics Agent, which is checked by default. But if you do NOT want to capture system metrics on the host machine you can uncheck the **Metrics Agent** box. See "About the .NET System Metrics Agent" on page 202 for more information.

Probe Aggregator. It is recommended that you install the Probe Aggregator Service, which is checked by default.

If you are installing the agent to work in an HP SaaS environment this option is required for SaaS and cannot be changed.

This Probe Aggregator service aggregates .NET Agent data to 5 second intervals before sending the performance data to the Diagnostics mediator server. This can improve scalability by reducing network communications with the server but the aggregator will also increase probe system overhead.See "Probe Aggregator Service" on page 44 for more information on the performance tradeoffs to installing the Probe Aggregator.

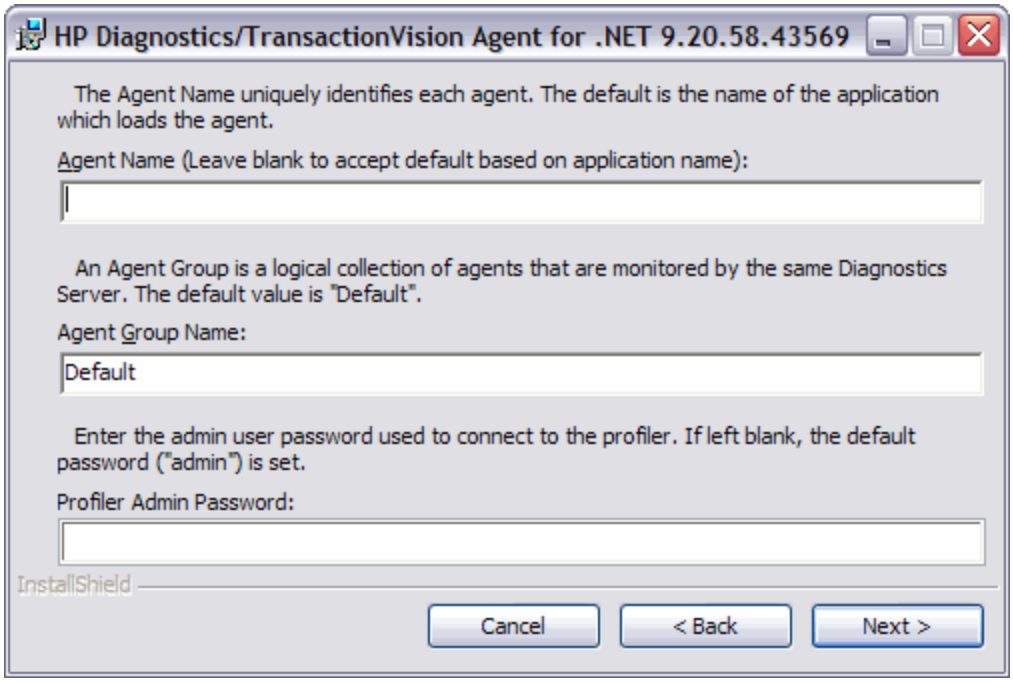
Disk Cost. To check the amount of available disk space on the drives of the host, click the **Disk Cost** button. Use this functionality to make sure that there is enough room for the Agent installation.

Click **Next** to proceed and continue to the next step.

Step 6. Agent name and group

Skip this step if the agent won't be reporting to a Diagnostics Server.

Enter the Agent Name and Agent Group Name.



Agent Name. The name that identifies the agent within HP Diagnostics. If you leave this field blank, the .NET Agent will auto-generate an agent name based on the application domain name of the monitored application. The agent name is assigned as the probe entity name.

> **Note:** It is recommended that you leave **Agent Name** blank and allow the agent to auto-generate the agent name. Read the following information carefully if you decide to enter your own agent name.

Note that Diagnostics does not support localization of agent names.

Considerations when entering an agent name:

- Valid characters that can appear in the agent name are: letters, digits, dashes, underscores, and periods.
- Assign an agent name that will help you recognize the application that is being monitored, and the type of instrumentation.

  For example, the agent name for the .NET Agent installed to monitor the application named `PetWorld` can be:

  ```
  PetWorld_Dotnet_Agent
  ```

- When you specify an agent name, all of the agents on the host are forced to use the same agent name.

The default agent name auto-generated by the agent when the agent name field is left blank is equivalent to specifying `$(MACHINENAME)_$(APPDOMAIN).NET`.

To override the default name, use the following substitution macros to enhance the name at run time:

- `$(MACHINENAME)`: Machine's host name
- `$(APPDOMAIN)`: Application's domain name
- `$(PID)`: Application's process ID
- `$(WEBSITENAME)`: The IIS Web site under which the application is hosted.
- $(COMMANDLINE:n) Where n is the command line parameter number.

  For example: +

  <id probeid="ILTEST_$(COMMANDLINE:3)_rest" probegroup="Default"/>

  with a command line of `iltest "heart and lung" -abc server` results in a probeid of ILTEST_server_rest.

Note that n=0 indicates the executable/command name.

> **Note:** For applications that are not hosted in IIS the agent name will be reverted to the default, that is, $(MACHINENAME)_$(APPDOMAIN).NET. An example of this would be console applications.
>
> For newly installed IIS applications you may need to run **Rescan ASP.NET Applications**or **Run HP .NET AppScanner** from the HP Diagnostics .NET Agent program group in the Windows Start menu.

Agent Group Name: Enter a name for an existing group or for a new group to be created. The default value for the agent group name is `Default.` The agent group name is case-sensitive. In Diagnostics this name is used as the probe group name.

Probe groups are logical groupings of probes that report to the same Diagnostics Server. The performance metrics for a probe group are tracked, and can be displayed on many of the Diagnostics views.

For example, you could assign all of the probes for a particular enterprise application to a single probe group so that you can monitor both the performance at the group level and the performance based on individual probe entities.

Profiler Admin Password. Enter the admin user password used to connect to the .NET Diagnostics Profiler. If left blank, the default password (admin) is set.

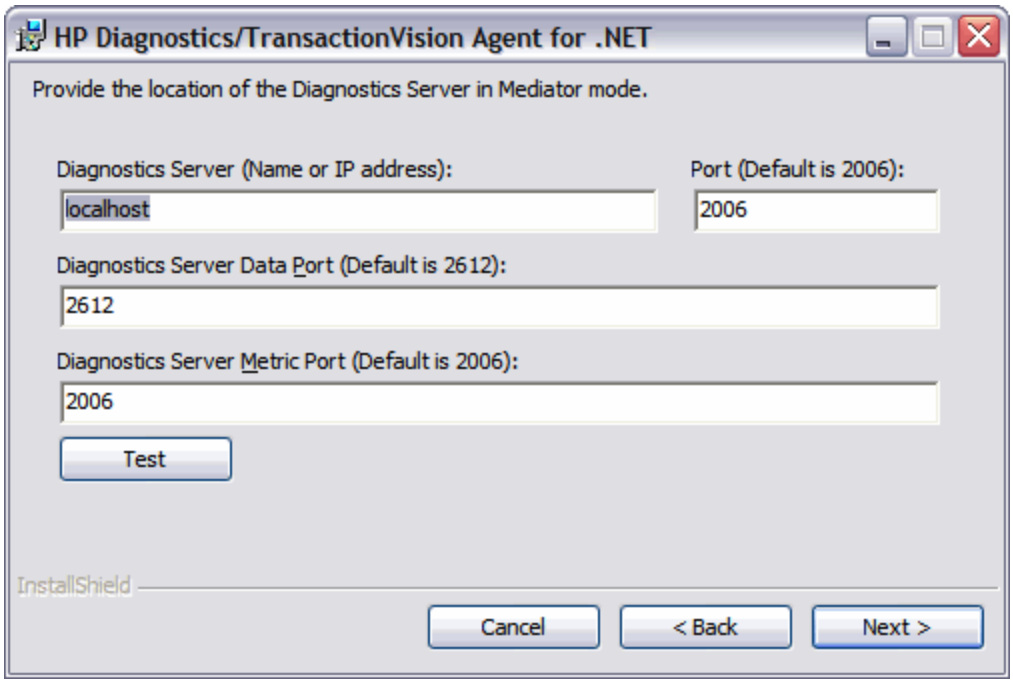
Click **Next** to proceed and continue to the next step.

Step 7. Diagnostics server information

Skip this step if the agent won't be reporting to a Diagnostics Server or if you are installing the agent to work in an HP SaaS environment. Your HP SaaS administrator will provide details for configuring communication between the agent and the SaaS-hosted Diagnostics Server.

Provide the information needed to enable the .NET Agent to communicate with the Diagnostics Server in Mediator mode.

If you selected to install the Probe Aggregator Service, you will see the Probe Aggregator Data Port instead of the Diagnostics Server Data Port and Probe Aggregator Metric Port instead of Diagnostics Server Metric Port.



- In the **Diagnostics Server (Name or IP address)** box, type the host name or IP address of the host for the Diagnostics Server in Mediator mode.

  Specify the fully qualified host name, not just the simple host name. In a mixed OS environment, where UNIX is one of the systems, this is essential for proper network routing.

- In the **Diagnostics Server Data Port** box, type the port number where the Diagnostics Server is listening for Agent communication. The default port number is 2612. If you changed the port since the Diagnostics Server was installed, specify that port number here instead of using the default.

  If you selected to install the Probe Aggregator Service, you will see the **Probe Aggregator Data Port** box instead of for the Diagnostics Server data port. Type in the port number where the Diagnostics mediator server is listening for the Agent communication when probe aggregation is installed. The default port number is 2626. If you changed the port since the Diagnostics Server was installed, specify that port number instead of using the default.

- In the **Diagnostics Server Metric Port** box, type the port number where the Diagnostics Server is listening for communications from the System Metrics Agent. The default port number is **2006**. If you changed the port since the Diagnostics Server was installed, specify that port number here instead of the default.

  If you selected to install the Probe Aggregator Service, you will see the **Probe Aggregator Metric Port** box instead of for the Diagnostics Server metric port. Type in the port number where the Diagnostics mediator server is listening for the Agent communication when probe aggregation is installed. The default port number is 45000. If you changed the port since the Diagnostics Server was installed, specify that port number instead of using the default.

- To perform a connectivity check to make sure that the Diagnostics Server is running and accessible from the installation host, click **Test**.

  The connectivity check lets you know right away if you made an error in the information you provided about the Diagnostics Server in Mediator mode, or if there is a connection problem between the Diagnostics Server's host and the Agent's host. If the connection to the Diagnostics Server in Mediator mode host cannot be resolved, an error message is displayed.

Click **Next** to proceed and continue to the next step.

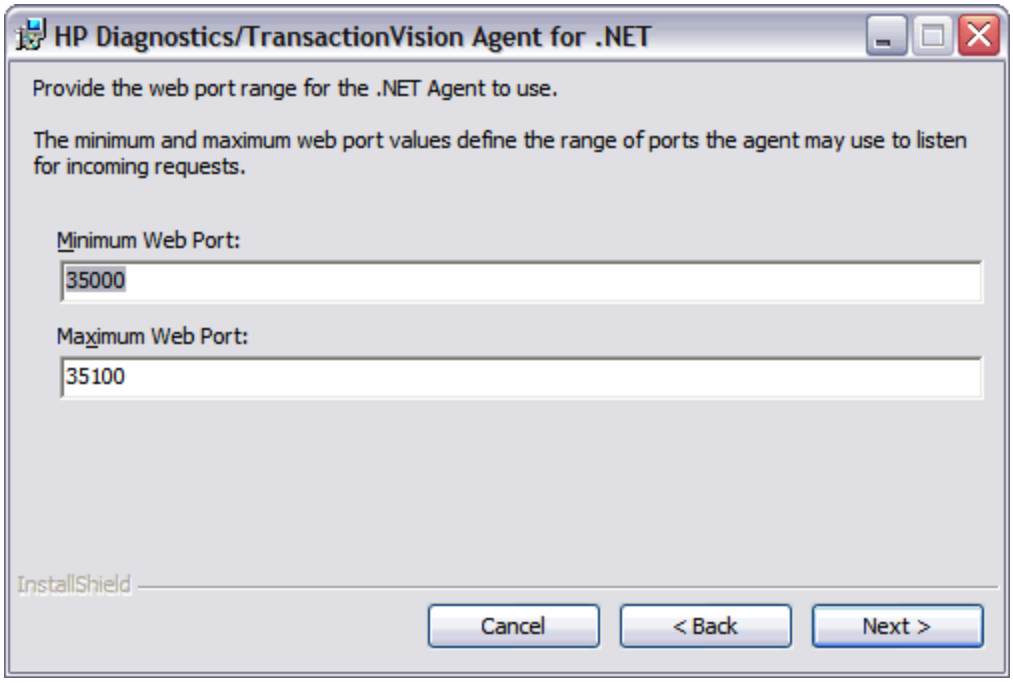
Step 8. Port and connection information

You will see different port and connection configuration dialogs depending on what install options you selected. Select from the following and proceed with the configuration:

- Port connection information for Diagnostics Servers
- Port and connection information for TransactionVision Server
- Profiler mode with no connection to a Diagnostics or TransactionVision Server

If you are installing the Agent to work with a Diagnostics Server, you will see the following dialog box.

Provide the Web port range for the .NET Agent to use.



- **Minimum Web Port.** Type the lowest port number, in a range of ports on the Agent host, you want to assign to the Agent.
- **Maximum Web Port.** Type the highest port number, in a range of ports on the Agent host, you want to assign to the Agent.

The default range is from 35000 to 35100 (inclusive).

The upper and lower limits of the Web Port Range are defined by the **Minimum Web Port** and **Maximum Web Port** fields. The Web Port Range contains the ports the Agent can use.

When an Agent is started, it attempts to find an unused port from within this range, starting from the lowest port number in the range and working its way up to the highest. Ports within the range could already be in use if another Agent or application previously claimed them.

The minimum size for the port range is equal to the maximum number of Agents that will be concurrently running on the Agent's host.

Considerations when setting the Web Port Range:

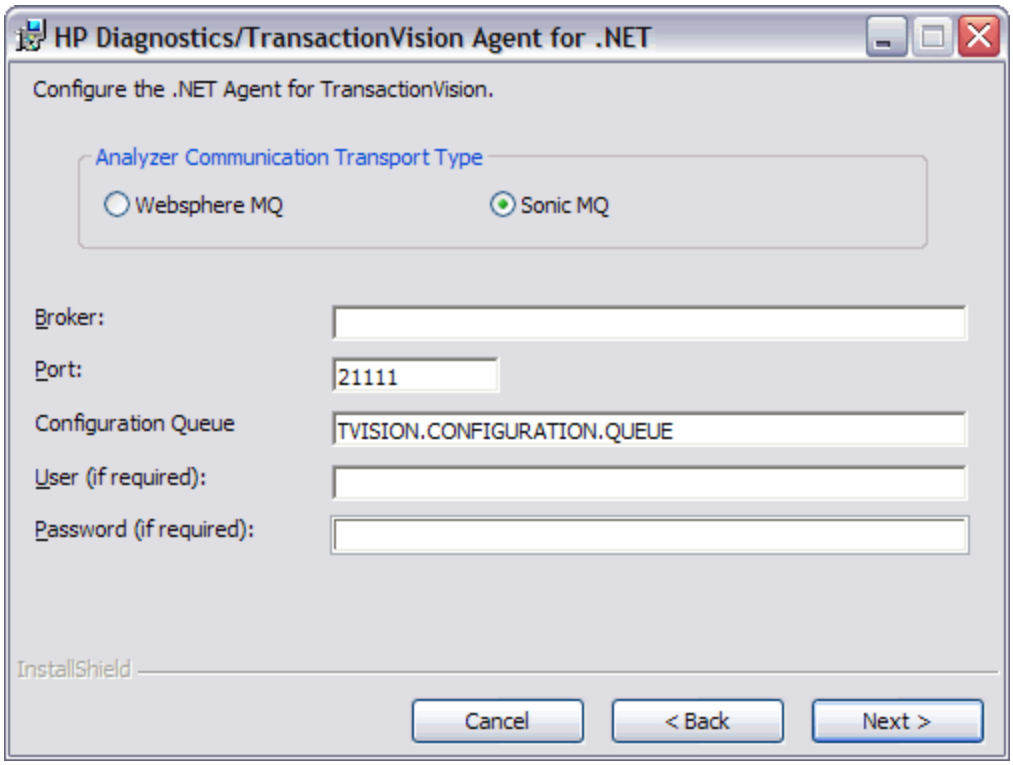
- If the Agents are working with ASP.NET applications, double the number of ports to account for ASP.NET's AppDomain recycling.
- If you have a firewall between the Agent and a component that will be communicating with the Agent, open the firewall for the ports within the range. Adjust the range to be just big enough.

Click **Next** to proceed and continue to the next step. If you also selected the option to have the agent work in a TransactionVision Environment see the following section for additional configuration.

If you are installing the Agent to work in a TransactionVision environment, you will see the following dialog box.

If you selected to install the agent to work with a TransactionVision Server you will see additional screens in the installation. See the *TransactionVision Deployment Guide* for information on using the agent in a TransactionVision environment.

The Configure the .NET Agent for TransactionVision dialog box appears.



Choose the Messaging Middleware Provider. Options are: WebSphere MQ and SonicMQ.

SonicMQ is included with the .NET Agent. If you specify this option, the Sonic MQ .NET client (Sonic.Client.dll - Progress SonicMQ .NET Client, version 7.6.0.112) is installed as part of the Agent installation.

A third-party WebSphere MQ installation can be used instead. In this case, you must install the MQ series .NET client (amqmdnet.dll - WebSphere MQ Classes for .NET, version 1.0.0.3) on the host being monitored.

By default, SonicMQ is selected.

- For SonicMQ, enter the following:

  Broker. Host name on which the Sonic broker is running. Typically this will be the Analyzer hostname.

  Port. The port on which the broker communicates. By default, 21111.

Configuration Queue. Name of the configuration queue. By default, TVISION.CONFIGURATION.QUEUE.

User. User id if required by SonicMQ installation for connection. By default, no username is required.

Password. Password if required by SonicMQ installation for connection. This is in the obfuscated form created by using the **PassGen** utility. By default, no password is required. For more information about **PassGen**, see "Command-Line Utilities" in Using Transaction Management.

- For WebSphere MQ, enter the following:

  Host. The host on which the WebSphere MQ queue manager resides.

  Port. Port number for WebSphere MQ queue manager.

  Configuration Queue. Name of the configuration queue.

  User. User id if required by WebSphere installation for connection.

  Password. Password if required by the WebSphere MQ installation for connection. This is in the obfuscated form created by using the PassGen utility. For more information about **PassGen**, see "Command-Line Utilities" in Using Transaction Management.

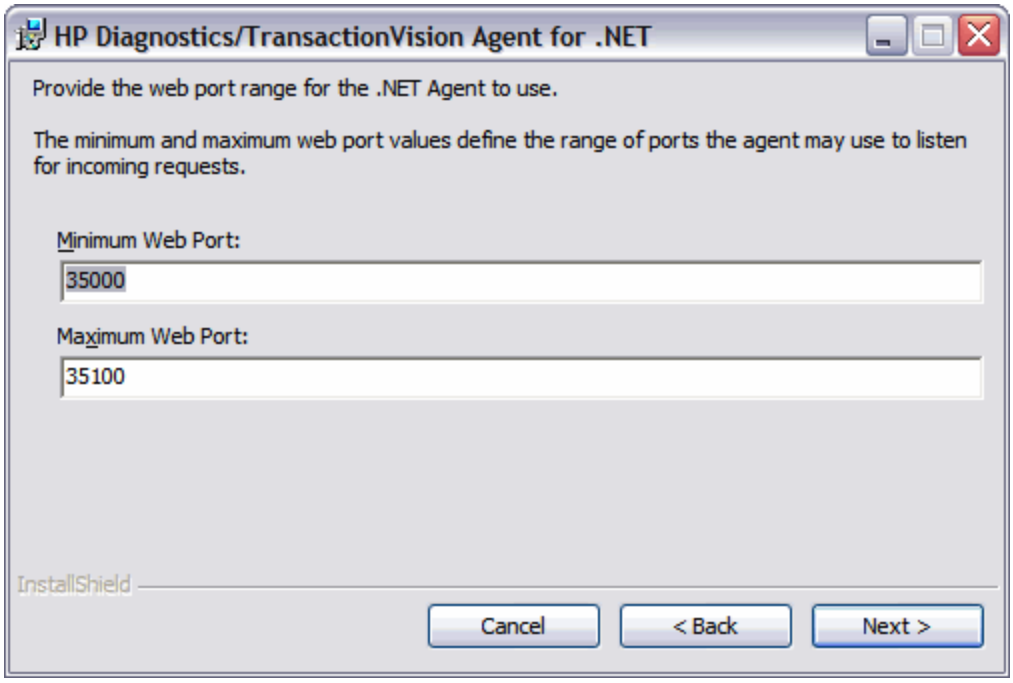
  Websphere MQ channel. Channel name for WebSphere MQ queue manager.

  Websphere MQ Q Manager. CCSID for WebSphere.

Click **Next** to proceed and continue to the next step.

If you are installing the Agent in Profiler mode, you will see the following dialog box:

Provide the Web port range for the .NET Agent to use.



Minimum Web Port. Type the lowest port number, in a range of ports on the Agent host, you want to assign to the Agent.

Maximum Web Port. Type the highest port number, in a range of ports on the Agent host, you want to assign to the Agent.

Note: The default range is from 35000 to 35100 (inclusive).

The upper and lower limits of the Web Port Range are defined by the **Minimum Web Port** and **Maximum Web Port** fields. The Web Port Range contains the ports that the Agent can use.

When an Agent is started, it attempts to find an unused port from within this range; starting from the lowest port number in the range and working its way up to the highest. Ports within the range could already be in use if another Agent or application previously claimed them.

The minimum size for the port range is equal to the maximum number of Agents that will be concurrently running on the Agent's host.

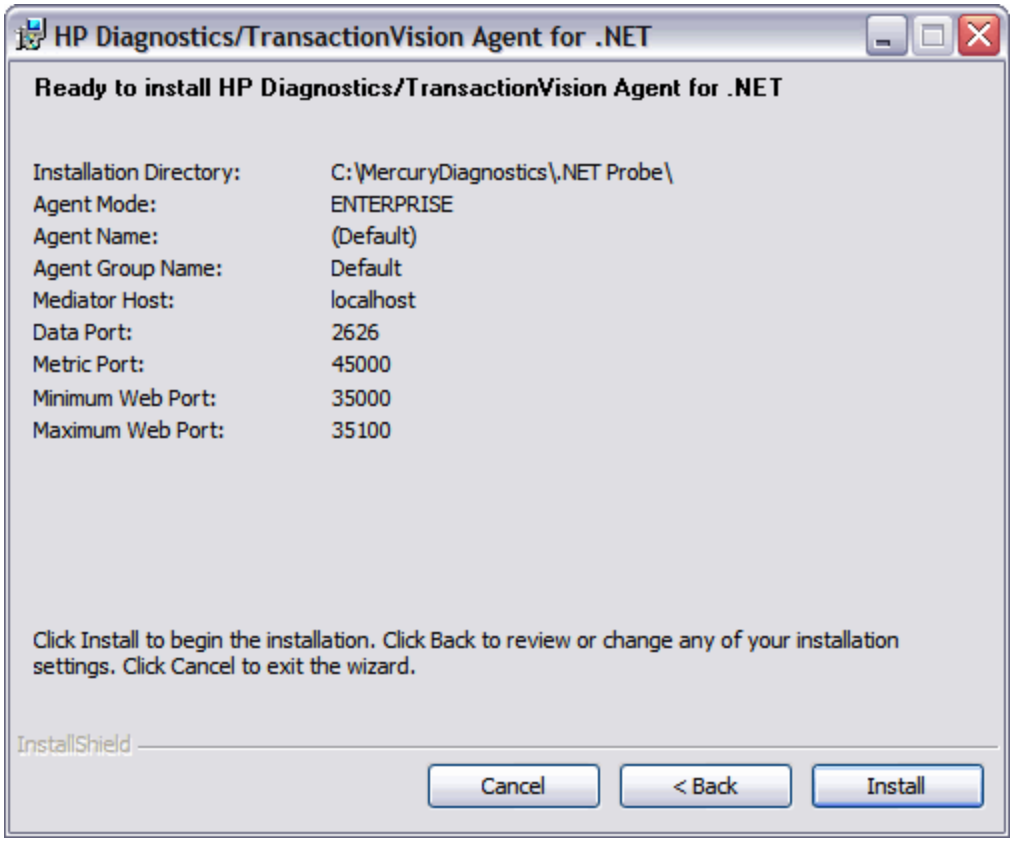
Considerations when setting the Web Port Range:

- If the Agents are working with ASP.NET applications, it is recommended that you double the number of ports to account for ASP.NET'sAppDomain recycling.
- If you have a firewall between the Agent and a component that will be communicating with the Agent, you must open the firewall for the ports within the range. For this reason you might want to adjust the range to be just big enough.

Click **Next** to proceed and continue to the next step.

Step 9. Pre-install summary

The pre-installation summary screen opens. Click **Back** to make any changes. Click **Install** to start the .NET Agent installation.



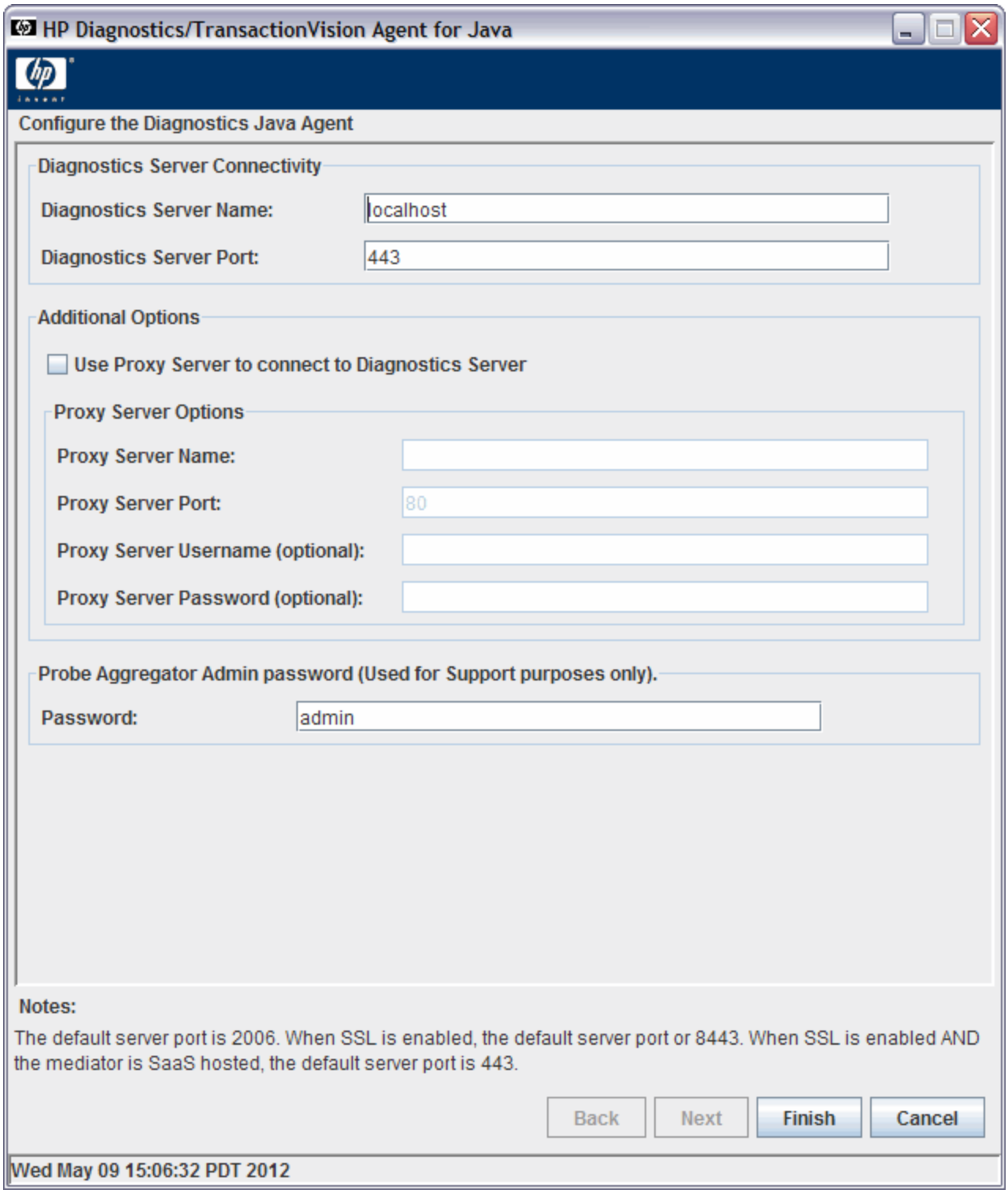
Note: When installing the agent for use as a Profiler only, there is no test for Metric Port connectivity.

If you are installing the agent to work in an HP SaaS environment continue to Step 10 otherwise skip the next step and continue to Step 11.

Step 10. Additional Setup for Agents Working in an HP SaaS Environment

If you are installing the agent to work in an HP SaaS environment then the SaaS Setup module starts automatically or you can run the SaaS Setup module anytime by selecting **Start > All Programs > HP Diagnostics .NET Probe > SaaS Setup**.

In the SaaS Setup module the following dialog is displayed. If you are not setting up the agent for an HP SaaS environment then you will not see this dialog.



- **Diagnostics Server Connectivity.** In an HP SaaS environment the Diagnostics Server is setup by HP on a system on-premise at HP. The default port for a SaaS environment is **443**. An HP SaaS administrator will provide you with the information on the Diagnostics Server host name and port to

use.

- If a proxy server is used to communicate with the Diagnostics Mediator Server select **Use Proxy Server to connect to Diagnostics Server** check box and enter the appropriate options. In an HP SaaS environment if your company requires a proxy to communicate to outside servers then you would select this option.
  - **Proxy Server Name.** Host name of the proxy server.
  - **Proxy Server Port.** Port of the proxy server.
  - **Proxy Server Username (optional).** The user used to authenticate the proxy server.
  - **Proxy Server Password (optional).** The password used to authenticate the proxy server.

  Proxy Server Options:

- **Probe Aggregator Admin password.** The password is automatically set to the same password as the .NET Profiler Admin password (entered in step 5), so for an initial agent setup for SaaS you will not see this field. If you want to subsequently change the Probe Aggregator Admin password, you can run the SaaS Setup module again and this field will be displayed.

Continue on to the next step to finish the installation.

Step 11. Post Install Information

On the final installation screen, you can select the Show the Windows Installer Log checkbox to view the log file and check for errors.

Click **Finish** to exit the installer.

For information on post installation tasks see "Post Install Tasks" on the next page.

When you are ready you must restart IIS, see the next step.

Step 12. Restart IIS

Restart IIS or the Web publishing service to pick up the new agent configuration.

- To restart IIS from the command line or from the **Start > Run** menu, type **iisreset** and press **Enter**.
- To restart the Web publishing service, use the Service Control Manager on Windows (%windir%\system32\services.msc).

For Diagnostics these commands restart the Web publishing service but do not immediately start the.NET Agent. The next time that a Web page in the application is requested, the agent is started, the applications are instrumented, and the agent registers with the Diagnostics Server.

For TransactionVision these commands restart the Web publishing service but does not immediately start the .NET Agent. The next time that a Web page in the application is requested, the agent is started, the applications are instrumented, and the agent reads the Configuration Queue Messages from the Analyzer.

Note: ASP.NET automatically restarts applications under various circumstances, including when it detects that applications are redeployed, or when applications exceed the configured resource thresholds.

When ASP.NET restarts an application that is being monitored by a .NET Agent, the agent is deactivated and a new agent is started. While this is occurring, there can be a period of overlap where there are multiple agents simultaneously registered with the Diagnostics Server in Commander mode and connected to the Diagnostics Server in Mediator mode. This condition could cause LoadRunner / Performance Center and BSM to report errors during the application restart sequence.

Continue with the next section to learn more about post installation tasks.

For information on verifying the installation see "Verifying the .NET Agent Installation" below.

Post Install Tasks

See the following topics for information about additional configuration for the .NET Agent:

- For information on how the .NET Agent automatically discovers applications and configures standard instrumentation to allow monitoring see "Discovery and Standard Instrumentation" on page 41.
- For information on configuring the .NET Agent for Diagnostics and for links to more advanced topics see "About Configuration of the .NET Agent for Diagnostics" on the next page.
- For information on configuring the .NET Agent for TransactionVision and to see the types of events TransactionVision can trace with the .NET Agent see "About Configuration of the .NET Agent for TransactionVision" on the next page.
- "Enabling and Disabling Standard Instrumentation for Applications" on page 48 for more information.
- For information on configuration for environments with proxies or firewalls, see the "Configuring for HTTP Proxy and Firewalls" chapter in the HP Diagnostics Installation and Configuration Guide.
- For information on enabling HTTPS, see the "Enabling HTTPS Between Components" chapter in the HP Diagnostics Installation and Configuration Guide.

Verifying the .NET Agent Installation

On the final installation screen you can select the **Show the Windows Installer Log** checkbox to view the log file and check for errors.

Log files are created in **<probe_install_dir>/log**. A log file is created for each discovered AppDomain.

The .NET probe does not register with the Diagnostics Server until the probe is started. The probe is started and registered with the Server when the instrumented application is run. For ASP.NET applications this happens the first time a page is requested for the instrumented application.

Once a .NET probe is started you can launch the Diagnostics Enterprise UI to verify that the probe is working. Access the System Health view to see details about each .NET probe and the machines that host them. See "How to Access the Diagnostics UI" in the Diagnostics Help system or the HP Diagnostics User Guide.

About Configuration of the .NET Agent for Diagnostics

You can customize the .NET Agent configuration and add custom instrumentation to suit your environment and the performance issues you would like to diagnose.

The installer configures your ASP.NET applications and the .NET Agent to work together to capture the basic workload of the applications. It is possible that one or more of your ASP.NET applications was deployed in a manner that prevents the installer from detecting it. Or, you might want to enhance the standard instrumentation to capture the performance metrics for the custom classes in the application.

In Diagnostics, you can do additional configuration using the **probe_config.xml** file. For details on this file see "Understanding the .NET Agent Configuration File " on page 85. For instructions on advanced .NET Agent configuration, see "Advanced .NET Agent Configuration" on page 170.

Also in Diagnostics, you can create custom instrumentation points to handle unique situations in your application environment. For general information on custom instrumentation see "Custom Instrumentation for .NET Applications" on page 59.

About Configuration of the .NET Agent for TransactionVision

When used with TransactionVision the .NET Agent captures events from .NET applications and sends the events to the TransactionVision Analyzer. See the *BSM Documentation Library* for more information about TransactionVision.

.NET Agent Configuration for TransactionVision

The default configuration of the .NET Agent allows you to begin tracing certain .NET events in a monitored application. You can customize the .NET Agent configuration to control what .NET events are traced and sent to the TransactionVision Analyzer.

To override the default configuration, access the **<agent_install_dir>/etc/ probe_config.xml** file. See "Understanding the .NET Agent Configuration File " on page 85 for details on the elements you can configure for both Diagnostics and TransactionVision.

The <modes> element in the **probe_config.xml** file is set at installation for both Diagnostics and TransactionVision (see "<modes> element" on page 133).

When you select to install the .NET Agent to work in a TransactionVision environment the <modes> element in the **probe_config.xml** file is set to tv. When this is the only mode selected the agent will

work in a TV only mode where the Profiler and the Diagnostics probe is disabled and only TV events are generated. When you select to install the .NET Agent to work in other modes such as with Diagnostics then both TV events and Diagnostics data collection will be enabled.

In order to specify TransactionVision specific and TransactionVision transport specific configuration the following elements in the **probe_config.xml** file are used exclusively for TransactionVision:

- <tv> element (see "<tv> element" on page 161 for details)
- <timeskew> element (see "<timeskew> element" on page 156 for details)
- <transport> element (see "<transport> element" on page 158 for details)
- <gentvhttpeventforwcf> element (see "<httpheaderrule> element" on page 112 for details)

If the .NET Agent is using SonicMQ transport to communicate with the TransactionVision Analyzer, SSL can be enabled. See the *BSM Hardening Guide* for details.

Types of Events TransactionVision Can Trace with the .NET Agent

When used with TransactionVision the .NET Agent traces the following types of .NET events:

1. Web Services
   a. ASP.NET (*.asmx) - Client and Server

      To generate events, use the **ASP.NET.points** file.

   b. WCF (*.svc) - Client and Server

      To generate events, use the **wcf.points** file.

   c. REST style services - Server

      To generate events, use the **wcf.points** file and set up the instrumentation of the application as described in "Configure WCF REST Services for Monitoring" on page 72.

2. Database calls executed using ADO.NET

   To generate events, use the **ADO.points** file.

3. .NET Remoting - Client and Server

   To generate .NET remoting events, use the **Remoting.points** file and setup the application for instrumentation as described in "How to Configure Instrumentation for .NET Remoting" on page 77.

4. MSMQ - Send and Receive (asynchronous)

   To generate events, use the **Msmq.points** file.

5. HTTP

   a. Client outbound - includes calls to REST services

      To generate events, use the **ASP.NET.points** file.

   b. ASP.NET inbound/server (POST, GET, PUT) (*.aspx)

      To generate events for HTTP, use **ASP.NET.points** file.

6. User defined events

   Use the detail argument **tv:user_event** (see "Optional Point Entries" on page 63)

   To turn off event generation remove the relevant points file from scope.

Enabling Correlation of .NET Events

The following .NET correlation rules are available by default in the BSM Transaction Management user interface and can be enabled from the Event Customization Rules page. For details about correlation rules, see "Custom Correlation" in the BSM Application Administration Guide.

- .NETMSMQRule
- .NETRemotingRule
- .NETRule
- .NETWCFRule

SSL Configuration for TransactionVision .NET Agents

For TransctionVision if the .NET Agent is using SonicMQ for the messaging middleware, SSL can be enabled. See “Configur the .NET Agent to Use SSL” in the *HP TransactionVision Deployment Guide* for details. Also see the *BSM Hardening Guide*.

Discovery and Standard Instrumentation

The .NET Agent installer automatically discovers the ASP.NET applications you might want to instrument. After you install the .NET Agent, you can request that the agent rescan your IIS configuration to catch any additions or changes.

Discovering ASP.NET Applications During Installation

The .NET Agent installer detects ASP.NET applications on the machine when the agent is installed. The .NET Agent installer discovers applications by inspecting the IIS configuration and looking for virtual directory entries that might refer to ASP.NET applications.

In some instances, the ASP.NET applications are installed in a manner that prevents them from being detected. An example is when an ASP.NET application is installed as a Web directory instead of a virtual directory.

Discovering ASP.NET Applications After Installation

You can request a rescan of the IIS configuration if you modified an existing ASP.NET application deployment or installed new ASP.NET applications. You can use one of the following options:

- To rescan the IIS configuration and automatically update the **probe_config.xml** file, select **Start > All Programs > HP Diagnostics .NET Probe > Rescan ASP.NET Applications**.
- To rescan the IIS configuration and manually select the applications and services to be monitored, **All Programs > HP Diagnostics .NET Probe > Run HP .NET AppScanner**. For details of this option, see "Manually Enabling Auto-Discovered ASP.NET Applications and Non ASP.NET Services" on page 201.

Automatic Instrumentation and Configuration for Discovered ASP.NET Applications

The .NET Agent installer configures the agent to capture basic ASP.NET/ADO/WCF workload for each of the ASP.NET applications detected. The agent performs the following configuration steps:

- Creates an application-specific capture points file template.

  The capture points file defines the instrumentation that controls the workload that the agent captures for each application. You can modify the instrumentation in the capture points file to provide instructions that allow the agent to capture performance data for application-specific custom methods. See "About Instrumentation and Capture Points Files" on page 59.

- Creates an **<appdomain>** tag in the **probe_config.xml** file, which is located in the **<probe_install_dir>/etc** directory. The attributes of the **<appdomain>** tag direct the behavior of the .NET Agent (points and enabled attributes). See "Understanding the .NET Agent Configuration File " on page 85 for details.

> **Note:** Diagnostics enables the instrumentation for all discovered applications by setting the **enablealldomains** attribute in the **process** tag to "true", which overrides the appdomain tag's enabled attribute. For information on enabling and disabling instrumentation for applications see "Disabling Logging" on page 192.

Population of BSM's RTSM

Diagnostics populates CIs and model relationships in the BSM Run-time Service Model (RTSM) for application infrastructure elements and business transactions.

For CI population the .NET Agent installer automatically discovers the IIS configuration metadata for ASP.NET applications that are deployed under IIS versions 6.x or greater. The discovered IIS configuration metadata is written to the **iis_discovery_data.xml** file which is located in the <**probe_install_dir>\etc** directory. After you have installed the .NET Agent, you can request that the agent re-scan your IIS configuration to update for any additions or changes.

- Runtime Population CIs for IIS Deployed ASP.NET Applications

  At runtime the .NET Agent queries the **iis_discovery_data.xml** file for IIS configuration metadata associated with the instrumented AppDomain. If the associated metadata is found, the agent

forwards the data to its Diagnostic Server which populates the BSM Run-time Service Model CIs for .NET Application. See integration with the BSM Run-time Service Model model for .NET Applications.

- Discovery of IIS Metadata of IIS Deployed ASP.NET Applications During Installation

  The .NET Agent installer discovers IIS deployed ASP.NET applications on the machine when the agent is installed. The .NET Agent installer discovers applications by querying the WMI (WMEB) Provider for the IIS configuration metadata. The pertinent metadata is written to the **iis_discovery_data.xml** file.

- Discovery of IIS Metadata of IIS Deployed ASP.NET Applications After Installation

  You must request a re-scan of the IIS configuration metadata when you have modified an existing ASP.NET application deployment or installed new ASP.NET applications. To request that the agent re-scan the IIS configuration and write a new **iis_discovery_data.xml** file, run **Start > HP Diagnostics .NET Probe > Rescan ASP.NET Applications** shortcut. Note that the new **iis_discovery_data.xml** file is not intended for editing by the user; any such user edits will be overwritten by executing this shortcut.

- Privilege Requirements for Discovery of IIS Deployed ASP.NET Applications

  The user must have Administrator privileges on the machine that the .NET Agent is installed on, otherwise the WMI queries will fail and the **iis_discovery_data.xml** file will not be created.

- Debugging the Discovery of IIS Deployed ASP.NET Applications

  If the **iis_discovery_data.xml** file is not created or there is any reason to suspect that some of its metadata may be inaccurate, you can enable the creation of a detailed debug file to examine the results of the WMI queries. To enable the creation of a detailed debug file. change last parameter of the Target Property for the **Start > HP Diagnostics .NET Probe > Rescan ASP.NET Applications** shortcut from "false" to "true". When the Rescan ASP.NET Applications shortcut is executed, an **<probe_install_dir>/log/AutoDetect.log** is created. Note that you should have Administrator privileges when executing this shortcut. You can send the **AutoDetect.log** to HP Support for analysis.

For information about setting up the integration with BSM, see the BSM-Diagnostics Integration Guide.

Non ASP.NET Applications

The .NET Agent installation automatically discovers your ASP.NET applications, creates settings for the applications in the **probe_config.xml**, and creates template points file for them. For each non-ASP.NET application—for example, NT Service, console application, UI client—you must create the appropriate settings in the **probe_config.xml** settings to configure the .NET Agent to monitor your applications as well as create points files indicating which points in your application you want to monitor.

The following is an example of a **probe_config.xml** setting for an application called **SimpleConsoleHost.exe**:

```
<process name="SimpleConsoleHost">
  <points file="SimpleConsoleHost.points"/>
  <logging level=" "/>
</process>
```

The following is an example of points file setting for an application called SimpleConsoleHost.exe:

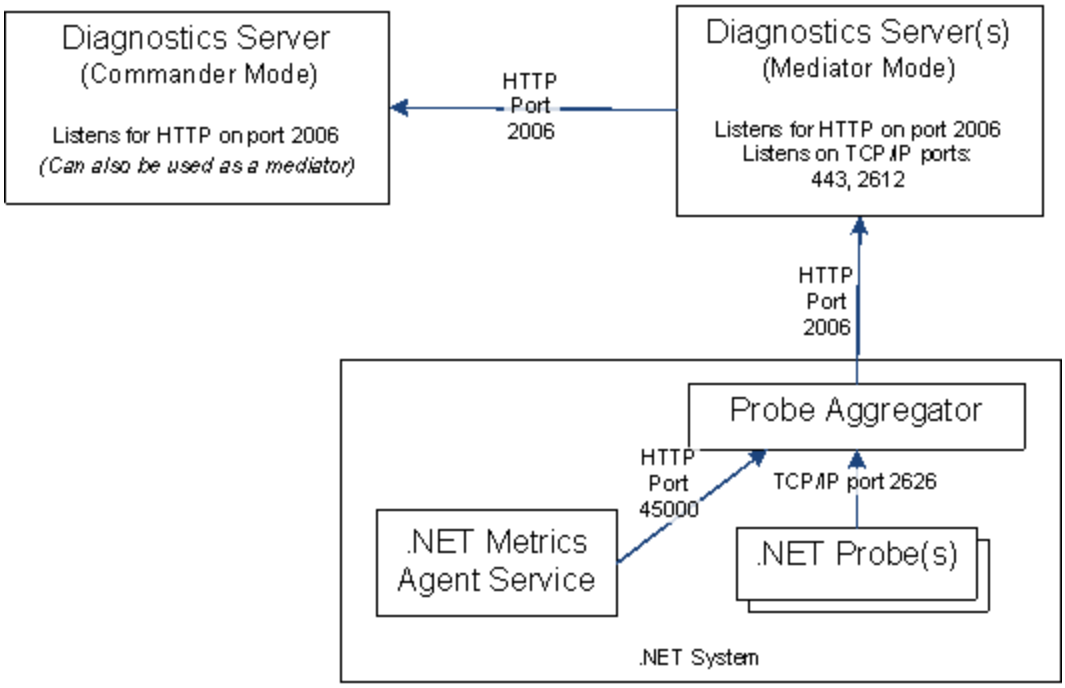
```
[SimpleConsoleHost]
class = MyNamespace.SimpleConsoleHost
method = !.*
ignoreMethod = Main
layer = SimpleConsoleHost
```

See "Custom Instrumentation for .NET Applications" on page 59 for more details.

Note: To monitor services, use the .NET Application Scanner utility. For details, see "Manually Enabling Auto-Discovered ASP.NET Applications and Non ASP.NET Services" on page 201.

Probe Aggregator Service

The Probe Aggregator Service can optionally be installed as part of the .NET Agent installation. It runs as a Windows Service, **HP Probe Aggregator**.



The Probe Aggregator Service aggregates probe data to 5 second intervals before sending the performance data to the Diagnostics mediator server. This is useful when the volume of data collected based on instrumentation of multiple applications is high and networking traffic would be too great if not aggregated.

The basic .NET Agent installation, without the Probe Aggregator Service, results in performance data being sent to the Diagnostics mediator server as method starts and stops occur.

There are performance trade-offs to using the Probe Aggregator Service. So you must assess the requirements in your environment. For example, consider using the probe aggregator when you have two or more .NET probe instances running on the same system. Actual network overhead is dependent on the applications being monitored, so you need to determine if the potential savings in network bandwidth and mediator load offsets the increased memory usage on the application system.

When you install the .NET Agent with the Probe Aggregator Service, this service runs automatically and waits for connections from the .NET probes. Standard configuration of the probe aggregator is done during the .NET Agent installation. The **<probe_install_dir>\ProbeAggregator\etc\probeaggregator.properties** file is used to set configuration parameters for the Probe Aggregator (for example, setting the SQL trending threshold).

If you decide, post installation, to install the Probe Aggregator Service you can run the .NET Agent installation again, selecting the **Change** button. Then select the check box for installing the **Probe Aggregator Service**.

Uninstalling the .NET Agent also removes the Probe Aggregator Service.

See "Enabling and Disabling the Diagnostics Agent for .NET" on page 48 for how to disable and enable the Probe Aggregator Service.

Monitoring NET Applications Deployed in Azure Cloud

Microsoft provides Windows Azure SDK for developers to create and deploy Azure applications to the Microsoft Windows Azure Cloud Infrastructure. The Diagnostics .NET Agent leverages the Azure SDK to provide seamless deployment of the .NET Agent into the Azure Infrastructure. Once deployed the .NET Agent monitors applications running in the Azure Cloud, collecting performance data and reporting to an HP Diagnostics Server for analysis and problem detection. See the **AzurePackReadMe.pdf** in the .NET Agent AzurePack zip file for details on installing and configuring the .NET Agent for monitoring applications in the Windows Azure Cloud.

Monitoring Applications on SharePoint with the .NET Agent

SharePoint is a web application that runs on ASP.NET and therefore the .NET Agent monitors it like any other ASP.NET-based web application. For instance, the .NET agent collects metrics that allow you to see:

- **Web services**. All calls to Web services that are serviced in the monitored SharePoint environment, or any Web services that are called from within the SharePoint environment, are captured.

- **Server Requests**. All incoming HTTP server requests to the SharePoint Server are captured.
- **SQL statements**. All outgoing database calls made in the SharePoint applications are captured.

You can perform additional configurations to further support monitoring of SharePoint by the .NET Agent as described below.

Note that SharePoints sites with virtual directories of the same name (AppDomain name) are distinguished by the full IIS path in the AppDomain\Probe Name and can be configured separately in the **probe_config.xml** file.

- Monitor SharePoint Web Parts with custom instrumentation by discovering points using the Reflector.

  See "Discovering the Classes and Methods in an Application" on page 175.

- Monitor the SharePoint SQL Server with a Diagnostics Collector. SharePoint Servers typically use one or more instances of SQL Server databases to store configuration and data. Install and configure a Collector to monitor each instance of these databases.

  See the HP Diagnostics Collector Guide.

- Monitor SharePoint performance counters at the host level. By default, the NET system metrics agent collects some Perfmon counters that are expected to be useful for SharePoint monitoring. You can add additional Perfmon counters.

  See "Adding System Metrics Using the Windows Performance Monitor" on page 205.

- Monitor SharePoint performance counters at the probe level. Configure AppDomain-specific metrics using the using the <metrics> and <metric> elements in the <probe_install_dir>\etc\probe_ config.xm. file.

  See ".NET Agent Configuration Elements" on page 86.

- Distinguish different SharePoint team sites with similar URLs by specifying key arguments in the <httpcaptureparams> element.

  See ".NET Agent Configuration Elements" on page 86.

- Consolidate "layout" server requests in SharePoint by specifying the <urireplacepattern> element. For example, this pattern specifies everything that is fetching layouts gets into one server request:

```
<symbols>
    <urireplacepattern enabled="true">
      <pattern value="s#(?i)(^.*)(_layouts).*$#Layouts#" />
    </urireplacepattern>
</symbols
```

This configuration is especially useful with newer versions of SharePoint, such as 2010 and 2013, where the default instrumentation results in numerous server requests.

For another example, this pattern consolidates all pages of the same name by stripping out the path.

```
<symbols>
    <urireplacepattern enabled="true">
         <pattern value="s#(?i)(^.*)(?&lt;word1&gt;/.*\.(aspx|asmx|ashx)$)
#${word1}" />
    </urireplacepattern>
</symbols>
```

This configuration changes two URIs such as these:

```
/div/20rpo/r3-r8_ops/4.101_afa/4.101.001_listmap/blog/Lists/Links/AllItems.aspx

/About/Directorates/PublishingImages/Forms/AllItems.aspx
```

To this:

```
/AllItems.aspx
```

- Adjust or configure automatic URI collapsing as needed for your monitoring requirements by using the <uriautocollapsing> element. By default, this feature is enabled.

  See ".NET Agent Configuration Elements" on page 86.

- Use the $(MACHINENAME), $(COMMANDLINE:2), and $(WEBSITENAME) macros for probe naming.

  SharePoint web sites often have names that include numbers and GUIDs. Assign more meaningful names to the probes by using macros for the probe name. See "Considerations when entering an agent name:" on page 27.

Collected performance data from SharePoint servers is displayed in the Microsoft SharePoint Server view group of the Diagnostics Enterprise UI. For details on the user interface, see the HP Diagnostics User Guide.

Determining the Version of the .NET Agent

When you request support, it is useful to know the version of the Diagnostics components you installed.

To determine the version of the .NET Agent:

- Right-click the file **<Agent_install_dir>\bin\HP.Profiler.dll** and select **Properties** from the menu. In the Properties dialog, select the Version tab to display the component version information.

or

- Use the System Health view in the Diagnostics UI.

Enabling and Disabling the Diagnostics Agent for .NET

The .NET Agent is enabled when it is installed. After you restart your Web server and a URL in the application is accessed, the .NET Agent begins to gather performance information.

You can disable the .NET Agent so that it does not start and does not gather performance metrics.

To disable a .NET Agent:

Select **Start > All Programs > HP Diagnostics .NET Probe > Disable HP .NET Probe**.

To enable a .NET Agent that was disabled:

Select **Start > All Programs > HP Diagnostics .NET Probe > Enable HP .NET Probe**.

> **Note:** Disabling the .NET Agent only disables the probe metrics collector and the active probes. It does not disable the system metrics collector. The process of enabling or disabling system metrics collection is controlled through the standard Windows services manager. The effect of enabling or disabling probes only happens the next time the probed application restarts. It has no affect on currently running applications.

Once the Probe Aggregator Service is installed and running, you can disable and enable it from the Start Menu. Select **Start > All Programs > HP Diagnostics .NET Probe > Disable HP .NET Probe** or Enable HP .NET Probe. Selecting Disable HP .NET Probe, in addition to disabling the .NET probes will mark the Probe Aggregator Service as disabled, but not stop the service (in case there are running probes remaining). Selecting Enable HP .NET Probe, in addition to enabling the .NET probes will change the Probe Aggregator Service back to type **automatic** and start it if needed.

Enabling and Disabling Standard Instrumentation for Applications

When the .NET Agent is first installed, the standard ASP.NET/ADO instrumentation for all discovered applications is enabled, but no application specific instrumentation is enabled. You control which applications have their instrumentation enabled or disabled using the attributes of the enablealldoamins attribute in the <process> element and attributes in the <appdomain> element in the **probe_ config.xml** file for the .NET Agent.

Disabling instrumentation for an application allows you to avoid the processing overhead and distracting information in the Diagnostics views for applications that are not relevant to the environment whose performance you want to monitor.

Enabling instrumentation for all application allows the .NET Agent to monitor the performance of all detected applications so that you can see the performance metrics for all of the applications in the views of the Diagnostics and Profiler user interfaces.

These are the rules for the enablealldomains attribute of the <process> element:

- enablealldomains = false : If there are no domains in the list of <appdomain> No domains should be enabled.
- enablealldomains = false : If there are domains in the list of <appdomain> Domains should be enabled if the "enable" attribute is set to true or not defined in the enable attribute of the <appdomain>.
- enablealldomains = true : If there are domains in the list of <appdomain> Only Domains in the list should be enabled disregarding their "enable" attribute.
- enablealldomains = true : If there are no domains in the list of <appdomain> All domains should be enabled.
- enablealldomains attribute is not defined: same as if enablealldomains = true.

To enable or disable the instrumentation for an application:

1. Set the **enablealldomains** attribute in the **<process>** element to **false**. This allows the attributes of each <**appdomain>** tag to control the state of the instrumentation for each application. If there are no <**appdomain>** entries, no applications are enabled.
2. Set the **enabled** attribute in the <**appdomain>** element to **true** for each application where you want to enable the instrumentation.
3. Set the **enabled** attribute in the **<appdomain>** element to **false** for each application that is to have its instrumentation disabled.

The following example shows instrumentation enabled for one application and disabled for another.

```
<process name="ASP.NET" enablealldomains="false">
  <points file="ASP.NET.points" />
  <points file="ADO.points" />
  <appdomain name="1/ROOT/myApplication" website=”Default Web Site”
enabled="true">
    <points file="DefaultWebsite-myApplication.points" />
  </appdomain>
  <appdomain name="1/ROOT/myApplicationTwo" website=”Default Web Site”
enabled="false">
    <points file="DefaultWebsite-myApplicationTwo.points"/>
  </appdomain>
</process>
```

To enable the instrumentation for ALL applications:

Set the **enablealldomains** attribute in the **<process>** element to **true**. This overrides the settings of the attributes in each **<appdomain>** element so that the instrumentation can be enabled without having to set numerous attributes.

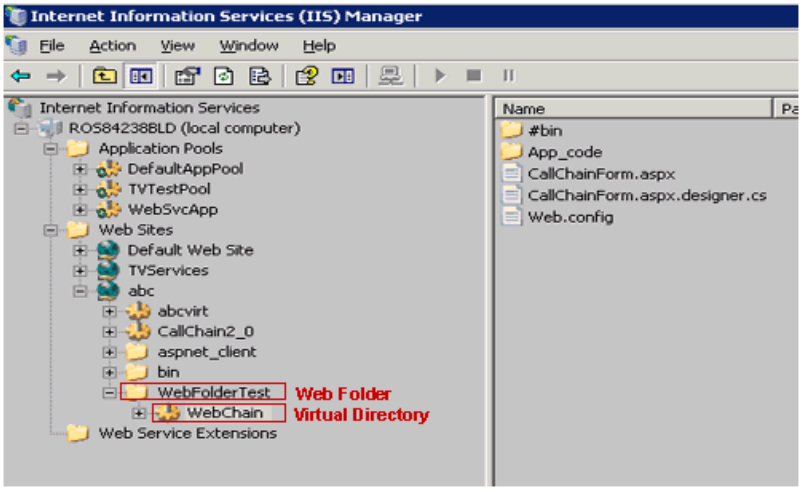
The following example shows instrumentation enabled for all applications:

```
<process name="ASP.NET" enablealldomains="true">
  <logging level=""/>
  <points file="ASP.NET.points"/>
  <appdomain name="1/ROOT/myApplication" website=”Default Web Site”
enabled="false">
    <points file="DefaultWebsite-myApplication.points"/>
  </appdomain>
  <appdomain name="1/ROOT/myApplicationTwo" website=”Default Web Site”
enabled="false">
    <points file="DefaultWebsite-myApplicationTwo.points"/>
  </appdomain>
</process>
```

Troubleshooting .NET Web Applications Not Discovered

In a Microsoft Windows 2003 server and IIS 6 environment, if your web site has a virtual directory under a web folder .NET Agent may fail to discover the virtual directory. This is because of an issue with the Microsoft WMI provider used by Diagnostics to walk down the web site tree. The WMI provider does not properly recognize the web folder as an IIS web directory and so Diagnostics can’t discover the virtual directory under the folder. See the example described below.

The example shows web folder WebFolderTest under the web site abc. Under this web folder there is a virtual directory WebChain.



Because of an issue with the WMI provider, the listing in WMI for this web site would not show the WebFolderTest/WebChain virtual directory. The .NET Agent uses the listing from the WMI provider to discover web applications. So in situations like this, the .NET Agent may not be able to discover virtual directories under a web folder.

Microsoft recommends modifying the metabase directly or using a simple script like the following to set the folder style using ADSI:

```
Set objRoot = GetObject("IIS://localhost/W3SVC/1/Root/WebFolderTest")
objRoot.KeyType = "IIsWebDirectory"
objRoot.SetInfo()
```

Instead of using a script you can manually configure the web folder as an application in IIS. Once this is done it can be reverted to a non-application but the property would now be set and Diagnostics would be able to discover the web application.

Another option is to manually add the excluded APPDOMAIN in the ASP.NET AppDomain list in the **probe_config.xml** file.

Manually Adding an AppDomain Not Discovered

If an AppDomain that you expected to be discovered by the .NET Agent was not discovered, rescan the IIS configuration. If the application was added after the .NET Agent was installed it may not have been discovered. See "Discovering ASP.NET Applications After Installation" on page 42 for details on rescanning.

If the AppDomain still does not appear, you can manually add the AppDomain. Choose the option below that suits your application.

- "Add all AppDomains Without Any Filtering" below
- "Add all AppDomains that Match a Specific Name in the Entire IIS configuration" on the next page
- "Add a Specific AppDomain in the IIS Configuration" on page 53

After you modify the configuration as described below, restart IIS or the Web publishing service to pick up the new agent configuration. See "Step 12. Restart IIS" on page 37.

Add all AppDomains Without Any Filtering

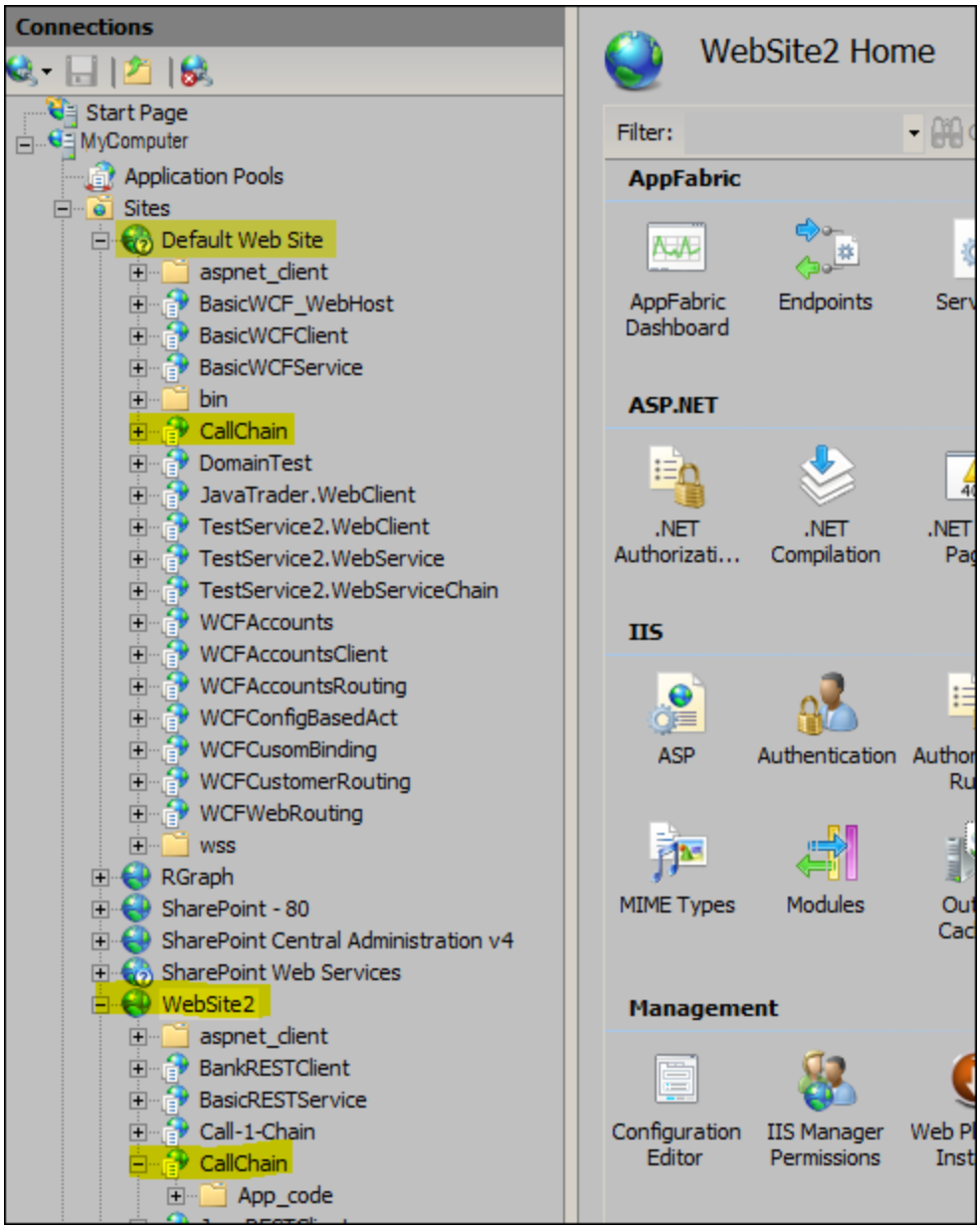
In the **<agent_install_dir>/etc/ probe_config.xml** file, locate the ASP.NET section and remove any existing <appdomain> elements. Then add the following section:

```
<process enablealldomains="true" name="ASP.NET">
  <logging level="" />
  <points file="ASP.NET.points" />
  <points file="ADO.points" />
  <points file="WCF.points" />
</process>
```

All AppDomains are enabled.

Add all AppDomains that Match a Specific Name in the Entire IIS configuration

Assume that you have multiple AppDomains of the same name, but in different web sites, to be included. For example the "CallChain" AppDomain below:



Add the entry shown in bold to the **<agent_install_dir>/etc/ probe_config.xml** file :

```
<process enablealldomains="false" name="ASP.NET">
 <logging level="" />
```

```
  <points file="ASP.NET.points" />
  <points file="ADO.points" />
  <points file="WCF.points" />
  <appdomain enabled="true" name="CallChain">
    <points file="CallChain.points" />
  </appdomain>
 </process>
```

All AppDomains of the same name are added, regardless of the web site in which they appear.

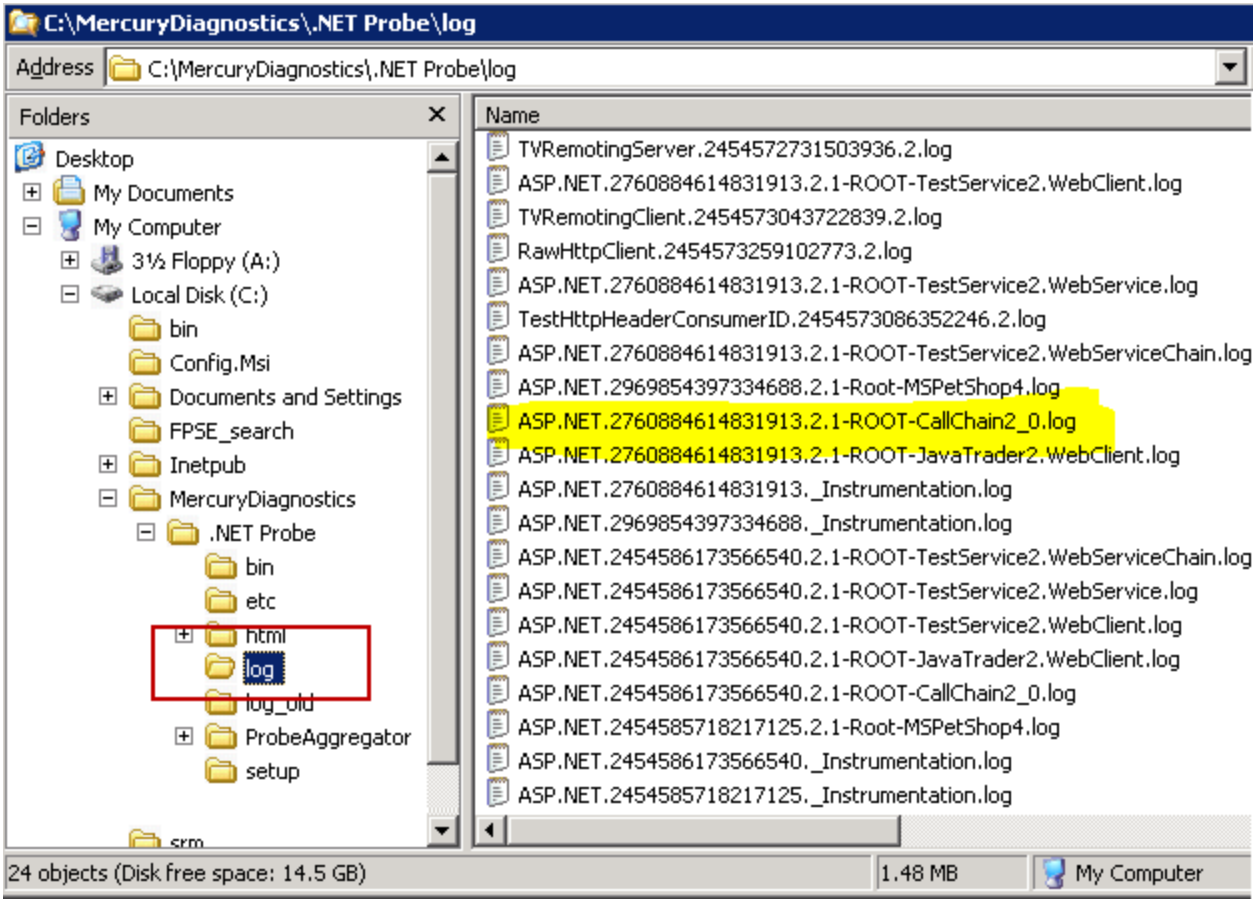
Add a Specific AppDomain in the IIS Configuration

Assume that you have multiple AppDomains of the same name as described in the previous example. To specify a particular AppDomain, specify the fully-qualified domain name as described below. For example, add the following to the **<agent_install_dir>/etc/ probe_config.xml** file to reference the CallChain AppDomain in WebSite2:

```
<process enablealldomains="false" name="ASP.NET">
 <logging level="" />
 <points file="ASP.NET.points" />
 <points file="ADO.points" />
 <points file="WCF.points" />
 <appdomain enabled="true" name="2/ROOT/CallChain" website="WebSite2">
   <points file="WebSite2-CallChain.points" />
 </appdomain>
```

To get the fully-qualified AppDomain name, perform the following steps.

1. In the **<probe_install_dir>\log** directory, locate the log file name that has the name of the virtual directory:



2. In the log file, locate an entry similar to the following:

```
2013.01.02.21.10.19.105  [0006]  INFO     AppDomain    Capture disabled for
appdomain(2/ROOT/CallChain) user(NT AUTHORITY\NETWORK SERVICE).
```

The highlighted name above is what should be used for the name value in the **probe_config.xml** file.

Other .NET Agent Troubleshooting Tips

If you have problems getting the agent started properly here are some things to check:

- Make sure you restarted the web server and that a URL in the application was accessed, this triggers the agent to begin collecting data.
- Check if a probe_config.xml file was created and is formatted correctly (that is, no missing tag closers, etc.). This can be done by opening the file in a web browser.

- Look for any message in the Windows Event Log named “HP Diagnostics”. This log is used exclusively by the .NET Agent. There should be a message for each attempt to instrument an application.
- After installing the .NET agent, Microsoft SharePoint 2013 may not function correctly. To fix this you can apply the following workaround:
  a. Open the SharePoint web.config file for editing. By default this file is located in **C:\inetpub\wwwroot\wss\VirtualDirectories\80**.
  b. Change the **legacyCasModel** setting from true to false, as follows:

```
<trust level="Full" originUrl="" legacyCasModel="false" />
```

  c. Restart IIS by using either IIS Manager or the IISReset command-line utility.

You can track the issue related to this workaround at http://blogs.msdn.com/b/jaskis/archive/2010/01/05/intermittently-getting-loading-this-assembly-would-produce-a-different-grant-set-from-other-instances-exception-from-hresult-0x80131401-after-net-3-5-sp-1.aspx.

Uninstalling the .NET Agent

To uninstall the .NET Agent:

1. Stop all Web applications that are using SOAP.
2. From the Windows Control Panel, select **Add/Remove Programs** and then select **HP Diagnostics/TransactionVision Agent for .NET** to uninstall.
3. Restart the Web applications.

   To remove the Probe Aggregator Service you can uninstall the .NET Agent which will also remove the Probe Aggregator Service. Or you can run the .NET Agent installation again, selecting the **Change** button and then de-select the check box for installing the **Probe Aggregator Service**.

Chapter 4: Upgrading the Diagnostics .NET Agent

This chapter presents the information you need to upgrade the Diagnostics .NET Agent.

This chapter includes:

- "Upgrade .NET Agents" below
- "Upgrade Notes and Limitations" on the next page

Upgrade .NET Agents

Consider the following when planning the Diagnostics Agent upgrade:

- You must upgrade the Diagnostics Server before upgrading the .NET Agents that are connected to it because Diagnostics Servers are not forward-compatible.

To upgrade a .NET Agent:

1. Install the new Diagnostics Agent for .NET (select Upgrade).

   The upgrade will take effect when the probed applications are restarted.

   To force the upgrade to take effect:

   a. Shut down all applications that are being monitored by the current .NET Probe.

   b. Restart IIS.

   c. Restart the applications that were being monitored by the old probe.

   See "Installing .NET Agents " on page 18 for additional information you need for installing a .NET Agent.

2. You can verify that the upgraded Diagnostics Agent is running by checking the version in the System Health view in the Diagnostics UI. The version should be the latest version if the upgrade was successful. To access the System Health view you must open the Diagnostics UI as the Mercury System customer from `http://<Diagnostics_Commanding_Server_Name>:2006/query/`. Then in the Views pane you can select the System Views view group.

Upgrade Notes and Limitations

In Diagnostics version 9.24, by default HTTP methods (such as PUT, GET, and POST) are used as an identifying component for each HTTP/S Server Request and a separate HTTP Server Request is generated for each HTTP method to the same URL. In earlier versions of Diagnostics, the root method of an HTTP Server Request is 'Server.Request' and one HTTP Server Request is generated for all HTTP methods to the same URL.

We recommend using the new method of Server Request identification, even though this is not backward compatible and breaks trend lines. If you must maintain continuity of trend lines, in the **probe_config.xml** file, change the value of the **symbols usehttpmethod** parameter to **false** (<symbols usehttpmethod="false" />).

Part 3: Advanced .NET Agent Configuration and Instrumentation

Chapter 5: Custom Instrumentation for .NET Applications

This section explains how to control the instrumentation that HP Diagnostics applies to the classes and methods of applications to enable the .NET Agent to gather the performance metrics.

This chapter includes:

- "About Instrumentation and Capture Points Files" below
- "Locating the .NET Capture Points Files" on the next page
- "Coding Points in the Capture Points File" on page 61
- "Instrumentation Examples" on page 66
- "Understanding the Overhead of Custom Instrumentation" on page 83
- "Default Layers for Typical .NET Applications" on page 83

About Instrumentation and Capture Points Files

Instrumentation refers to bytecode that the probe inserts into the class files of the application as they are loaded by the CLR. Instrumentation enables a probe to measure execution time, count invocations, and catch exceptions; and to correlate method calls and threads. The instrumentation points for each probe are specified in the capture points file.

The capture points file enables you to control the scope of the instrumentation so that Diagnostics can give you all the information you need to understand the performance of the applications without overwhelming you with costly or confusing extraneous information. The instrumentation definitions contained in the capture points file are called *points* that tell the probe which methods to instrument, how they should be instrumented, and which instrumentation should be installed.

Points can include regular expressions that "wildcard" the instructions so that they apply to more than one method, class or namespace specification. For more information about using regular expressions, see "Using Regular Expressions" in the HP Diagnostics User Guide.

You can customize the points in the capture point file to include methods, classes, and namespaces for areas of the application that do not fall within the default points.

The Microsoft specification for .NET does not include a unified or recommended interface that business logic should implement except in the case of instrumentation for web and WCF methods. This means that the .NET probe will almost always require custom points in the capture points file to enable it to gather meaningful metrics for the performance of the business logic classes and methods in .NET applications.

The points in the capture points file are grouped into layers. Layers organize the performance metrics into meaningful tiers of information that can be compared as part of a triage process and control the collection behavior of the instrumentation.

The points in the capture points files are grouped into default layers. You can customize the default layers and create new layers (see "Default Layers for Typical .NET Applications" on page 83).

Locating the .NET Capture Points Files

When you install the .NET Agent, predefined default capture points files are installed.

Default capture points files for ASP.NET applications are located at **<probe_install_dir>\etc** and include **Asp.Net.points, Ado.points** and **WCF.points** as well as other points files shown in the table below.

In addition, the .NET Agent installer automatically creates a separate capture points file for each IIS deployed ASP.NET Application Domain it detects. You must modify the automatically detected and created points file to enable custom instrumentation points for the Application Domain. These capture points files are located in the **<probe_install_dir>\etc\<ApplicationDomain>.points** file. These points files and the default points files are read by the .NET Agent.

At installation, only the **Asp.Net.points**, **Ado.points** and **WCF.points** default points files are *enabled*. The following default .NET points files are installed in the **<probe_install_dir>/etc** directory but *not enabled*:

Default Point File (initially disabled)	Instrumentation Target
Asp.Net.IExecutionStep.points	IIS5, IIS6 and IIS7. This file makes the IIS points obsolete.
IIS.points	IIS5 and IIS6
Lwmd.points	Lightweight Memory Diagnostics
Msmq.points	Microsoft Message Queuing (MSMQ instrumentation)
Remoting.points	.NET Remoting
WebServices.points	ASP.NET Web Services

You can enable the points files by adding a reference to them in the **<points>** element in the scope of the AppDomain in the **probe_config.xml** file. See "Understanding the .NET Agent Configuration File " on page 85 for details on each element in the **probe_config.xml** file.

For information on .NET probe instrumentation specific to TransactionVision, see the *HP TransactionVision Deployment Guide*.

Coding Points in the Capture Points File

The following arguments can be used to define a point in the points files:

```
[Point-Name] =<unique name for the point>
;-----------------------------------------------------------------------------
class = <class/package name/s to capture>
method = <method name/s to capture>
signature = <signature/s of method/s>
ignoreClass = <classes to ignore>
ignoreMethod= <method prototypes to ignore>
ignoreTree= <class hierarchy to ignore>
deep_mode= <soft or hard mode>
scope = <comma separated list of methods>
ignoreScope= <comma separated list of methods>
detail = <list of specifiers>
keyword = <keyword>
layer = <layer name>
layerType = <layer type>
```

Caution: Do not modify any of the default points files because, in an installation upgrade, modifications are lost. Store your application-specific instrumentation points in a custom capture points file.

All arguments that can be specified as a regular expression list have an effective maximum limit of 260 characters, which if exceeded results in a truncated value. The arguments are described in the following sections.

Mandatory Point Arguments

Every point, except for the points for LWMD, HttpCorrelation, WSCorrelation and WCF, must contain the following arguments:

Argument	Description
Point-Name	A unique name for the point.
class	Specifies the name of the class or interface to be instrumented. The name should include the full namespace name using periods between the namespace and class levels. Any valid regular expression can be used.

Argument	Description
method	Specifies the name of the method to be instrumented. To be successful, the method name must match a method defined in the class or interface specified by the class argument. Any valid regular expression can be used.
layer	Specifies a layer, sublayer, or tier under which the data from this point is grouped. Layers are a part of the instrumentation collection control. Layers in a point can be specified with nested layers or sublayers by separating the layer names with a / (slash). The layer specified following the slash is a sublayer of the layer specified before the slash. A sublayer can have its own sublayers by coding another slash and layer name following a sublayer name.

The following is an example of a custom point that contains the mandatory arguments:

```
[MyCustomEntry_1]
; comments here….
class = myNameSpace.myClass.MyFoo
method = myMethod
layer = myCustomStuff
```

Note: Regular expressions can be used for most of the arguments in a point. They must be prefaced with an exclamation point. For more information about using regular expressions, see "Using Regular Expressions" in the HP Diagnostics User Guide.

Optional Point Entries

Point definitions can contain one or more of the following arguments:

Argument	Description
keyword	Indicates special instrumentation. The keyword argument can be used to enable specific features; for example, the WCF keyword turns on the WCF feature. The keyword argument can also relate point definitions to special functionality; an example of this is the RemotingServer keyword and the Remoting.points file. • **HttpCorrelation.** Turns on correlation of client/server method calls via HTTP • **WsCorrelation.** Turns on web service correlation logic on the client side and turns on correlation of raw HTTP client request calls across both the .NET and Java technologies. • **WCF.** Turns on the WCF feature. • **REST.** Turns on the WCF REST service instrumentation. • **lwmd.** Turns on lwmd instrumentation. • **Remoting.** Turns on .NET Remoting framework instrumentation. • **RemotingServer.** Associates points in a .NET Remoting server to special .NET Remoting logic for these points. See "How to Configure Instrumentation for .NET Remoting" on page 77.
ignoreClass	Specifies a comma-separated list of classes to ignore. Any class matching one of the classes specified with **ignoreClass** is not instrumented.
ignoreMethod	Specifies a comma-separated list of methods to ignore. Any method matching one of the methods specified with **ignoreMethod** is not instrumented.
ignoreTree	Ignores instrumenting any method that is implemented on a class that inherits from the specified class. Thus, an entire class hierarchy tree of methods would be ignored.

Argument	Description
deep_mode	Specifies how subclasses are handled. This argument accepts three values: • **none** - A value of none is identical to not specifying a **deep_mode** argument. It has no effect on how subclasses are handled. • **soft** - A value of soft requests that, for every class or interface matching the class, method, and signature entries, any subclasses or subinterfaces that also implement the matching method and signature should also be instrumented. Soft mode is typically used for points for interfaces. • **hard** - A value of hard requests that, for every class or interface matching the class, method, and signature entries, any subclasses or subinterfaces at any depth should have all their methods instrumented. Hard mode is typically used for special cases. **Caution:** Hard mode can lead to extensive instrumentation and very high probe overhead.
scope	Constrains the context in which instrumentation is performed. If specified, the inserted bytecode is caller side. Any valid regular expression can be used for the value of this argument. Scope values are expressed as a comma-separated list of method names.
ignoreScope	Excludes certain methods from those included in the scope specified by the scope argument. Any valid regular expression may be used for the value of this argument. ignoreScope values are expressed as a comma-separated list of method names.

Argument	Description
detail	Provides more specific capture instructions. For the following the string that is returned is displayed in the method's Argument field in the details pane of the Call Profile view. It is a comma-separated list of the following: • **args:n** – Captures all supported types of arguments for the method (s) that match. A value of 'n' captures all arguments. Or you can enter a value for n from 1 through 256. • **args:0** – Calls the ToString() on the current class instance or callee object. This is invalid for static methods. • ***args:1** – Marks (*) the argument as a key argument for the server requests if the method is a top-level request. The detail argument also takes the following value: • **tv:user_event** - Generates a TransactionVision event for the methods that match. As part of the TransactionVision event the parameters to the method are collected as the Request Payload and the return value is collected as the Response Payload. The values displayed are the ToString() values returned by the parameters or the return value objects. Note that all parameters and return values may not be collected. Provides extensive support for transaction tracing by enabling TransactionVision event generation from practically any given method in any .NET application. You specify the method on which you want a TransactionVision event generated. It is highly recommended that event generation is specified for one method at a time to avoid too many events and performance degradation in TransactionVision. Avoid using wild card specifications (but they are supported for convenience).
layerType	Specifies special handling for some instrumented methods and accepts these values: • **trended_method** – Identifies methods to be displayed in the Trended Methods view. • **sql** – Identifies methods used to capture SQL for the SQL views. These are set by HP Diagnostics and should not be modified.
signature	Specifies the signature (return and parameter types); for example, System.String(System.int32, System.String). Any valid regular expression can be used.

Instrumentation Examples

The following examples illustrate how you can customize the instrumentation of an application by creating and modifying the points in the capture points file.

This section includes:

- "Custom layer and sublayer" below
- "Wildcard method" below
- "Ignore Specified Methods" on the next page
- "Capture Methods for the Trended Methods View" on the next page
- "Capture Only a Specific Method In a Class" on the next page
- "Capture a Specific Method That Returns a String" on page 68
- "Caller Side Instrumentation" on page 68
- "Argument Capture" on page 69
- "Configure WCF REST Services for Monitoring" on page 72
- "Deep_mode Examples" on page 73
- "How to Configure and Set Up Points for Non-ASP.NET or Windows Applications" on page 74
- "How to Configure Instrumentation for .NET Remoting" on page 77

Custom layer and sublayer

The following point creates a custom sublayer called BAR within the layer called FOO for the method myMethod in myCompany.myFoo class:

```
[myCompany.myFoo_customLayer]
class = myCompany.myFoo
method = myMethod
layer = FOO/BAR
```

Wildcard method

The following point captures all methods in the MyCompany.MyFoo class:

```
[myCompany.myFoo_AllMethods]
class = myCompany.myFoo
```

```
method = !.*
layer = FOO/BAR
```

Ignore Specified Methods

The following point captures all methods in the MyCompany.MyFoo class except for the methods setHomeInterface and getHomeInterface:

```
[myCompany.myFoo_AllMethodsExcept]
class = myCompany.myFoo
method = !.*
ignoreMethod = setHomeInterface,getHomeInterface
layer = FOO/BAR
```

The following point captures all methods in the MyCompany namespace except for those contained in the MyCompany.logging class:

```
[myCompany_All_Methods_except_from_MyCompany_Logging]
class = !myCompany\..*
method = !.*
ignoreClass = MyCompany.logging
layer = FOO/BAR
```

Capture Methods for the Trended Methods View

The following point captures the required data to populate the Trended Methods View for the myMethod method:

```
[myCompany.myFoo_customLayer]
class = myCompany.myFoo
method = myMethod
layer = FOO/BAR
layertype = trended_method
```

Capture Only a Specific Method In a Class

The following point captures all non-static constructor methods for the MyCompany.MyFoo class:

```
[myCompany.myFoo_Constructor]
class = myCompany.myFoo
method = .ctor
layer = FOO/BAR
```

The following point captures all static constructor methods for the MyCompany.MyFoo class:

```
[myCompany.myFoo_Singleton]
class = myCompany.myFoo
method = .cctor
layer = FOO/BAR
```

The following point captures the setFoo method in the MyCompany.MyFoo class:

```
[myCompany.myFoo_setFoo]
class = myCompany.myFoo
method = setFoo
layer = FOO/BAR
```

The following point captures all methods in the MyCompany.MyFoo class whose name includes “set”:

```
[myCompany.myFoo_AllSets]
class = myCompany.myFoo
method = !.*set.*
layer = FOO/BAR
```

The following point captures all methods in the MyCompany namespace:

```
[myCompany_All_Methods]
class = !myCompany\..*
method = !.*
layer = FOO/BAR
```

Capture a Specific Method That Returns a String

The following point captures the getFoo method that returns a System.String in the MyCompany.MyFoo class:

```
[myCompany.myFoo_GetFoo_String]
class = myCompany.myFoo
method = getFoo
signature = !System.String\(.*
layer = FOO/BAR
```

Caller Side Instrumentation

By default, all the instrumentation in Diagnostics is Callee side instrumentation where the bytecode is placed within the method call. Caller side instrumentation refers to the process of placing bytecode for measurement around the call to the method to be instrumented, instead of within the method.

Caller side instrumentation allows for finer control of instrumentation placement, but can increase the application initialization time because each class specified in the scope must be checked for references to the class/method specified in the points.

The scope and ignoreScope arguments are used to specify what caller should be instrumented. The following two examples refer to Caller side instrumentation.

The following point captures all methods in the MyCompany namespace that are called from the MyCompany.logging class.

```
[myCompany_All_Methods_from_MyCompany_Logging]
class = !myCompany\..*
method = !.*
scope = !MyCompany.logging.*
layer = FOO/BAR
```

The ignoreScope argument is used to exclude certain classes and methods from those included in the scope specified in scope argument. The following point captures all methods in the MyCompany namespace that are called from the MyCompany.logging class except for those called from the myMethod method.

```
[myCompany_All_Methods_except_from_MyCompany_Logging]
class = !myCompany\..*
method = !.*
scope = !MyCompany.logging.*
ignoreScope = MyCompany.logging.myMethod
layer = FOO/BAR
```

Argument Capture

The arguments to be captured are specified in the detail key of a points file section.

The following example calls the ToString() method of the n-th argument. The string that is returned is displayed in the method's Argument field in the Call Profile view: `detail=args:1,...args:4, *args:3`

There are several special values to note:

- args:n – Captures all supported types of arguments for the method(s) that match. A value of 'n' captures all arguments. Or you can enter a value for n from 1 through 256.
- args:0 – Calls the ToString() method on the current class instance or callee object.
- Adding a * to the args element (*args:1) marks a key argument.

To see the arguments for each method call, do not specify a key argument. This is a way to get more detailed information on the captured instance tree and could help answer questions about why this instance is a MAX tree or what values were passed in when there was an exception.

To group server requests for a method by arguments, specify a key argument. The key arguments, aggregate server requests with distinct values. Arguments that have a large number of distinct values are not good candidates for key arguments because this will lead to unique server requests for every distinct value.

Note: Even if you have not specified argument capture, arguments are captured when a method in the call tree throws an exception. These arguments are displayed in the Call Profile view, in the Stack Trace section of the Exceptions detail pages. See the Call Profile View online help for more details.

The following argument capture example relates to the code shown below:

```
[ILTest]
class = !ILTest_NameSpace.ILTest_Class
method = methodWithParams
detail = args:0, *args:3, args:5, args:7
layer = myFunctionLayer
```

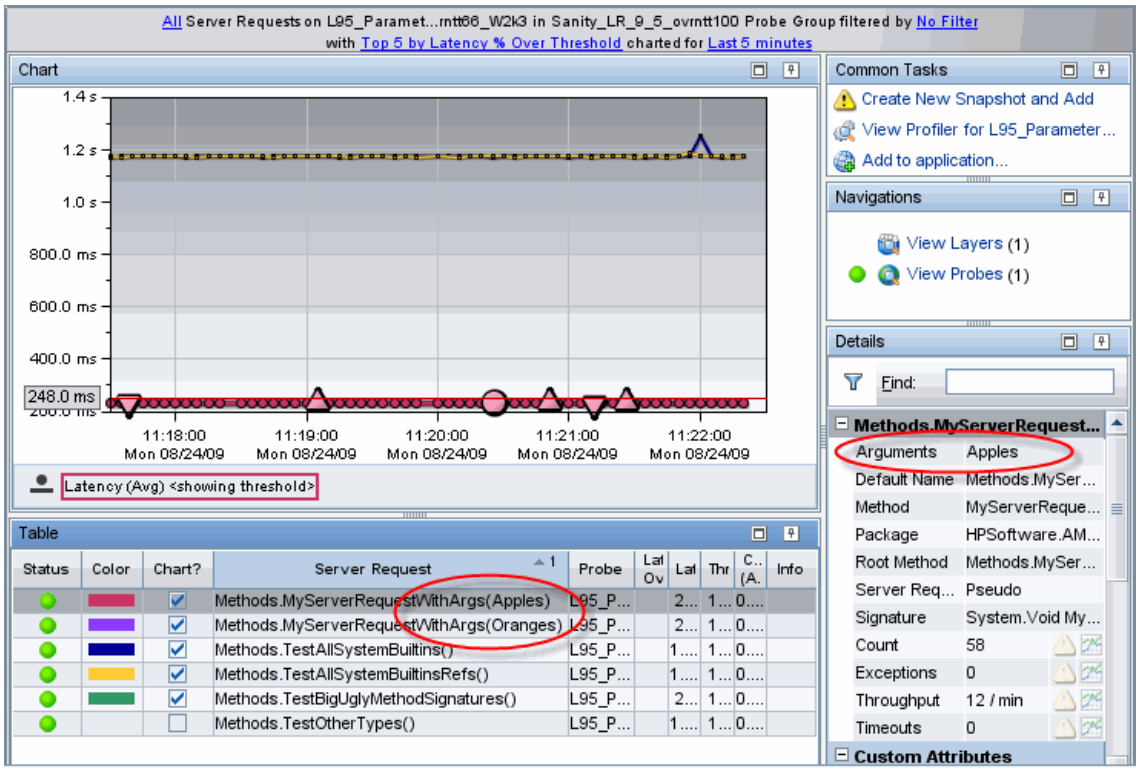
Here is the relevant code example:

```
class ILTest_Class
{
public bool methodWithParams
(string param1, int param2, string QnameParam3, long param4, object param5, int
param6, double param7)
{
... some implementation
}
}
In this example the defined detail will capture ILTest_Class.ToString(args:0)
param1, QnameParam3, param5 and
param7.
```

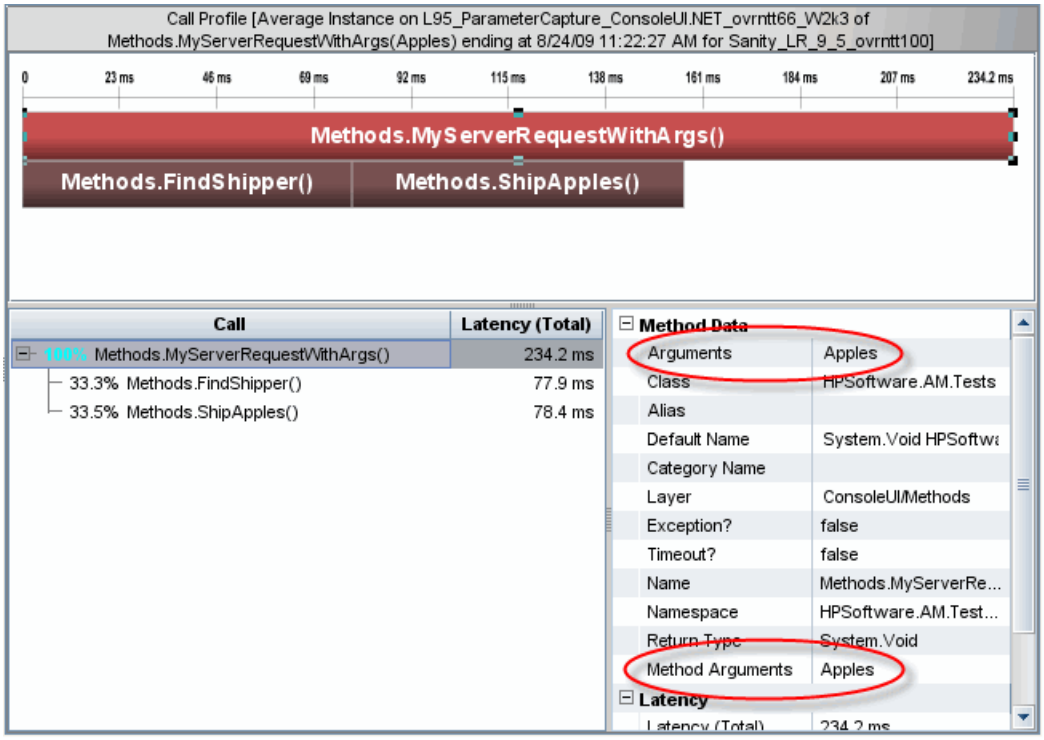
The value of QnameParam3 will be part of the identity of the server request if the top level method is methodWithParams.

When an argument to be captured is marked as a key argument (with an asterisk *) and the method is a top-level method, the argument value becomes part of the Server Request identity.

For example, if Shipping Type is a parameter of a method processing different shipments and you specify the Shipping Type argument as a key argument, you will be able to see aggregated views for each different shipment (apples and oranges) being processed by the method.



When you specify a key argument, the Call Profile view shows key arguments in the Arguments field in the Details pane. You will also see the arguments displayed under Method Arguments in the Details pane.



When arguments to be captured are NOT marked as key arguments (with no asterisk *), they are displayed in the Call Profile view under Method Arguments only.

Configure WCF REST Services for Monitoring

For a .NET Probe WCF REST services are monitored by default based on the **keyword=REST** value enabled out-of-the-box in the **WCF.points** file. These REST services will be monitored as web services and their performance data displayed in the Diagnostics UI SOA Services views.

You can further configure REST services as described in the sections below.

REST Service Configuration

In WCF REST style services sometimes the operations are encoded as url parameters. For example:

```
HTTP Method: PUT  Url: http://localhost:81/RestNOSvc/AccountsRESTService/
{ID}?op={OPERATION} op can be "deposit" or "withdrawal"
```

To be able to distinguish operations in these types of services you can specify the operation parameters of the REST service method as a **key argument** to allow it to be displayed as a separate operation. See "Argument Capture" on page 69 for a general description of argument capture.

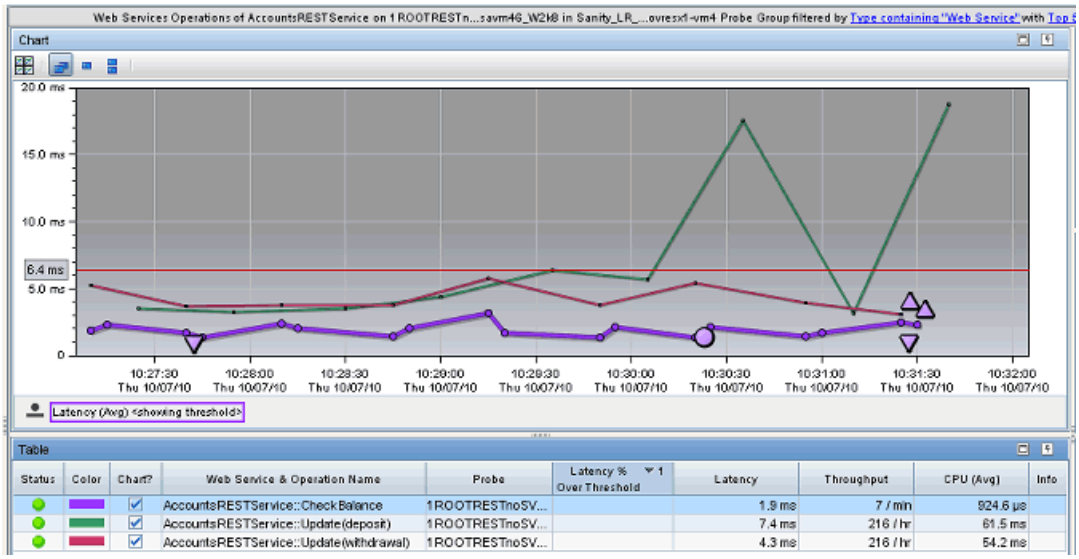
For example, for the method

```
[WebInvoke(UriTemplate = "{id}?op={operation}", Method = "PUT")]
public TransactionResult Update(string id, string operation, long Amount)
```

The operation is the key argument and can be specified in the points file as:

```
[WebSite2-RestNOSvc]
class = !HP.Test.WcfRestService.*
method = Update
detail = *args:2
layer = WebSite2-RestNOSvc
```

The SOA Services Operations view example below shows the results of this configuration with separate operations shown in the table.



REST Client Configuration

The REST service client is the same as an HTTP client call and cannot be distinguished. So for monitoring .NET applications that are REST service clients, the configuration option **<httpclient showurl="false"/>** should be set in the **probe_config.xml** file to avoid a large number of outbound calls and possible symbol table explosion. The number of calls is due to unique urls accessed by the client, often with ids encoded in the urls.

For example:

```
/RestNOSvc/AccountsRESTService/8FFD2F34-E334-4E1E-A940-50FCCCACE1D1
```

where the Guid represents different account ids.

Deep_mode Examples

The following interface definition is used for both soft and hard deep_mode examples:

```
public interface Interface1 {

  public void callerMethod();

}
```

The following class is used for both soft and hard deep_mode examples:

```
public class Class1 implements Interface1 {
 public void callerMethod(){
  calleeMethod();
  calleeMethod2();
 }

 public void calleeMethod(){
```

```
  Console.WriteLine("hello world");
  //more code lines here…
 }

 public void calleeMethod2(){
  Console.WriteLine("hello world 2");
 }
}
```

The following point captures the callerMethod in the Class1 class:

```
[Training-1]
class = Interface1
method = !.*
deep_mode = soft
layer = Training
```

The following point captures all methods in Class 1; that is, callerMethod, calleeMethod1, and calleeMethod2:

```
[Training-1]
class = Interface1
method = !.*
deep_mode = hard
layer = Training
```

How to Configure and Set Up Points for Non-ASP.NET or Windows Applications

This section explains how to configure both the **probe_config.xml** file and custom points files that enable instrumentation for Non-ASP.NET or Windows applications. Instrumentation for Windows Services, console applications, Windows Forms applications, and WPF applications are considered Windows applications and are referred to as such.

Windows Application Design

The critical point to consider when contemplating how to configure a Windows application you want to monitor is that the .NET probe is designed to monitor long running processes. Therefore, if your Windows application is designed to run for a few seconds and then exit, you will probably not be able to see any data for that run. When the Windows application exits quickly, the AppDomain is shut down and the probe is shut down before it can establish and maintain communication with a Diagnostics Server or the Diagnostics .NET Profiler.

The following simple Windows application illustrates a number of crucial concepts to be considered when configuring the instrumentation for a Windows application.

```
namespace Hello_dotNet_nameSpace
```

```
{
  class someclass
  {
    static void Main(string[] args)
    {
    // do something
    // read form commandline then exit
    clReader myClReader = new clReader();
    String cl;
    cl = myClReader.readCl();
    }
  }
  // Command Line Reader
  public class clReader
  {
    public String clread;
    public String readCl()
    {
      System.Console.WriteLine("Continue?");
      clread = Console.ReadLine();
      return clread;
    }
  }
}
```

The Hello_dotNet.exe Windows application has Main() that calls a method, waits for the user to enter something on the command line, and then exits. Until the application exits, the probe is active.

Creating the Hello_dotNet.points File

In the **<probe_install_dir>\bin** folder there is a **Reflector.exe** command line utility you can run against the Hello_dotNet.exe Windows application to obtain a suggested points file. See "Discovering the Classes and Methods in an Application" on page 175 for more information on the reflector utility.

When both the Reflector.exe and the Hello_dotNet.exe application are in the same folder, you would the following command:

```
Reflector.exe Hello_dotNet.exe
```

The output is sent to stdout. Among other information you will see the following suggested Hello_dotNet.points:

```
-----------------------------------------------------------------------
Sample .points by Namespace
-----------------------------------------------------------------------
[Hello_dotNet_nameSpace]
class = !Hello_dotNet_nameSpace.*
layer = Hello_dotNet_nameSpace
```

The suggested points can be used as is, except when the Windows application has a method like Main (); that is, a method that, if instrumented, does not return an exit until the application exits. In this case, the method spans the lifetime of the application so nothing would be reported until the application exits. Since the probe will be unloaded when the application exits, you will probably not get any data from the instrumentation point.

To fix this situation, construct a points file so that the Main() method, or any method like it, is not instrumented. The following Hello_dotNet.points file shows how to do this. It assumes that Main() is implemented in `someclass`.

Hello_dotNet.points:

```
[Hello_dotNet_nameSpace]
class = !Hello_dotNet_nameSpace.*
ignoreClass = Hello_dotNet_nameSpace.someclass
layer = Hello_dotNet_nameSpace

[ignore]
class = Hello_dotNet_nameSpace.someclass
ignoreMethod = Main
layer = Hello_dotNet_nameSpace
```

The crucial aspect of this type of points file is shown in bold. The [ignore] section instruments other methods in Hello_dotNet_nameSpace.someclass if there are any while ignoring the Main() method.

Configuring the Windows Application for Instrumentation

To configure the .NET probe to instrument the Hello_dotNet.exe Windows application, add the following XML to the **probe_config.xml** file. You can add it to the bottom of the file just above the **</probeconfig>** entry.

```
<process name="Hello_dotNet">
  <points file="Hello_dotNet.points" />
  <instrumentation>
    <logging level="" />
  </instrumentation>
  <logging level="" />
</process>
```

Note: You must place your **Hello_dotNet.points** file in the **<probe_install_dir>\etc** folder before you make the above changes to the **probe_config.xml** file.

The only required child element is the points file. The instrumentation, logging, and modes are optional. The following instrumentation setting can be useful when diagnosing which methods are or are not being instrumented:

```
<instrumentation>
  <logging level="points ilasm" />
</instrumentation>
```

How to Configure Instrumentation for .NET Remoting

You can configure the .NET probe to add custom instrumentation that supports the instrumentation of .NET Remoting Client and Server applications. Supported configurations are:

- Both HTTP and TCP bindings
- Both Binary and SOAP Formatting

Configuration

By default, the .NET probe is not enabled to instrument Remoting applications. You must add custom instrumentation points for both the Client and Server applications.

Two instrumentation keywords are related to Remoting:

Remoting. The Remoting keyword enables instrumentation for various points in the Remoting Framework.

RemotingServer. The RemotingServer keyword identifies the class that implements the Remoting Methods and isolates the instrumentation of the methods on that class from unintended instrumentation of other similar methods.

Client Example

The following very simple Windows application example illustrates a number of crucial concepts the must be considered when configuring the instrumentation for a Remoting Client Application.

```
namespace HPSoftware.AM.Tests.Remoting.SimpleRemoting
{
  class SimpleConsoleClient
  {
    [STAThread]
    static void Main(string[] args)
    {
      const string msg1 = "How are you?";
      String filename =
AppDomain.CurrentDomain.SetupInformation.ConfigurationFile;
      RemotingConfiguration.Configure(filename, false);
      MyRemotableObject remoteObject = new MyRemotableObject();
      doit(remoteObject, myMsg);
      Console.WriteLine();
      Console.WriteLine("(Press any key to exit)");
      Console.ReadKey();
    }
  public static void doit(MyRemotableObject obj, String message)
  {
    Console.WriteLine(obj.GetEnlightenment(message));
  }
}
```

As described in "How to Configure and Set Up Points for Non-ASP.NET or Windows Applications" on page 74, you can use the Reflector utility to help determine how to configure the Remoting Client points file.

To configure the probe to instrument the SimpleConsoleClient Remoting Windows application, add the following XML to the **probe_config.xml** file:

```
<process name="SimpleConsoleClient">
  <points file="Remoting.points" />
  <points file="SimpleConsoleClient.points" />
  <instrumentation><logging level="" /></instrumentation>
  <logging level="" />
</process>
```

You must add the **<points file="Remoting.points" />** entry.

If you are in the directory that holds the SimpleConsoleClient.exe and the Reflector.exe is in the PATH, you can execute the Reflector on the command line to view an implementation decomposition of the SimpleConsoleClient.exe and suggested point file settings:

```
Reflector SimpleConsoleClient.exe
```

The output of this command will contain the following:

```
Sample .points by Namespace
----------------------------------------------------------------------
[HPSoftware.AM.Tests.Remoting.SimpleRemoting]
class = !HPSoftware.AM.Tests.Remoting.SimpleRemoting.*
layer = HPSoftware/AM/Tests/Remoting/SimpleRemoting
----------------------------------------------------------------------
(1 classes) Namespace: HPSoftware.AM.Tests.Remoting.SimpleRemoting
----------------------------------------------------------------------
HPSoftware.AM.Tests.Remoting.SimpleRemoting.SimpleConsoleClient (8
Methods)
Equals                  System.Boolean(System.Object)
Finalize                System.Void()
GetHashCode             System.Int32()
GetType                 System.Type()
doit                    (method signature information unavailable))
Main                    System.Void(System.String[])
MemberwiseClone         System.Object()
ToString                System.String()
```

The suggested SimpleConsoleClient.points are:

```
[HPSoftware.AM.Tests.Remoting.SimpleRemoting]
class = !HPSoftware.AM.Tests.Remoting.SimpleRemoting.*
layer = HPSoftware/AM/Tests/Remoting/SimpleRemoting
```

These settings, however, would not create instrumentation that would produce any data. The reason, as discussed in "How to Configure and Set Up Points for Non-ASP.NET or Windows Applications" on

page 74, is that you must ignore methods like Main(). If you factor in the need to ignore Main(), you would be left with the following possible points file settings:

```
[HPSoftware.AM.Tests.Remoting.SimpleRemoting]
class = !HPSoftware.AM.Tests.Remoting.SimpleRemoting.*
ignoreMethod = Main
layer = HPSoftware/AM/Tests/Remoting/SimpleRemoting
```

Although these settings might be useful and would produce data, you should make them more precise. This is primarily due to probe performance. The more methods that are instrumented, the greater will be the probe's performance hit on the instrumented application. For example, if you can remove the wildcards "!.*" from the settings, the scope of your settings become explicit.

Notice from the Reflector output that there is actually only a single implemented class:

```
HPSoftware.AM.Tests.Remoting.SimpleRemoting.SimpleConsoleClient
```

You can remove the wildcards from the class setting as follows:

```
class = HPSoftware.AM.Tests.Remoting.SimpleRemoting.SimpleConsoleClient
```

Notice also, that the Reflector output does not contain a method setting. The default meaning of no method setting is that all methods are instrumented. Since most the following methods are only present because they are inherited from System.Object, it is unlikely that you really want to instrument these methods: `Equals, Finalize, GetHashCode, GetType, MemberwiseClone, ToString`. However, it is likely that you would want to instrument the `doit` method because it wraps the Remoting client call.

The following settings are recommended for the SimpleConsoleClient.points file:

```
[HPSoftware.AM.Tests.Remoting.SimpleRemoting]
class = HPSoftware.AM.Tests.Remoting.SimpleRemoting.SimpleConsoleClient
method = doit
layer = HPSoftware/AM/Tests/Remoting/SimpleRemoting
```

Server Example

The following Windows application example illustrates a number of crucial concepts the must be considered when configuring the instrumentation for a Remoting Server Application:

C# code snippets are shown for both the Remotable Object, which is shared between the Remoting Client and Server, and the SimpleConsoleServer.exe Remoting Server Application.

Here is the C# code snippet for the Remotable Object:

```
HPSoftware.AM.Tests.Remoting.SimpleRemoting
{
  public class MyRemotableObject : MarshalByRefObject
  {
    const string response = "I'm just fine!";
```

```
      public MyRemotableObject()
      {
      }
      public String GetEnlightenment(string message)
      {
        return response;
      }
    }
  }
```

Here is the C# code snippet for the SimpleConsoleServer.exe:

```
namespace HPSoftware.AM.Tests.Remoting.SimpleRemoting
{
  class SimpleConsoleServer
  {
    [STAThread]
    static void Main(string[] args)
    {
      String filename =
AppDomain.CurrentDomain.SetupInformation.ConfigurationFile;
      RemotingConfiguration.Configure(filename, false);

      Console.WriteLine("Server is running... press any key to exit");
      Console.ReadKey();
    }
  }
}
```

To configure the probe to instrument the SimpleConsoleServer Remoting WIndows application, add the following XML to the **probe_config.xml** file:

```
<process name="SimpleConsoleServer">
  <points file="SimpleConsoleServer.points" />
  <instrumentation><logging level="" /></instrumentation>
  <logging level="" />
</process>
```

You are not required to add the **<points file="Remoting.points" />** entry.

Point files for the Remoting Server can have one or more sections. The first section relates to the Remotable Object and is a required section. A second section that relates to the Remoting Server instrumentation can be added. Other optional sections can also be added to instrument other methods that can be called by either the Remoting methods or the Remoting Server. We will construct the Remotable Object section first.

The Remotable Object will reside in some assembly. We will assume it is in the RemotableObjects.dll.

When you run the Reflector against the RemotableObjects.dll, you see output that includes:

```
-----------------------------------------------------------------------
Sample .points by Namespace
-----------------------------------------------------------------------
[HPSoftware.AM.Tests.Remoting.SimpleRemoting]
class = !HPSoftware.AM.Tests.Remoting.SimpleRemoting.*
layer = HPSoftware/AM/Tests/Remoting/SimpleRemoting

-----------------------------------------------------------------------
(1 classes) Namespace: HPSoftware.AM.Tests.Remoting.SimpleRemoting
-----------------------------------------------------------------------
HPSoftware.AM.Tests.Remoting.SimpleRemoting.MyRemotableObject (17
Methods)
__RaceSetServerIdentity        System.Runtime.Remoting.ServerIden…)
__ResetServerIdentity          System.Void()
CanCastToXmlType               System.Boolean(System.String,System…)
CreateObjRef                   System.Runtime.Remoting.ObjRef(Syste…)
Equals                         System.Boolean(System.Object)
Finalize                       System.Void()
GetComIUnknown                 System.IntPtr(System.Boolean)
GetEnlightenment               System.String(System.String)
GetHashCode                    System.Int32()
GetLifetimeService             System.Object()
GetType                        System.Type()
InitializeLifetimeService      System.Object()
InvokeMember                   System.Object(System.String,System…)
IsInstanceOfType               System.Boolean(System.Type)
MemberwiseClone                System.MarshalByRefObject(System…)
MemberwiseClone                System.Object()
ToString                       System.String()
```

As with the Remoting Client example, you cannot just use the suggested point settings. You must be certain that you identified the class that implements the Remotable Object. You do this by observing that the Remotable Object is required to inherit from System.MarshalByRefObject and therefore must have the following methods on it: `CreateObjRef`, `GetLifetimeService`, `InitializeLifetimeService`, `MemberwiseClone`. From the Reflector output above, you can see that the HPSoftware.AM.Tests.Remoting.SimpleRemoting.MyRemotableObject class is an obvious candidate for the class that implements the Remotable Object.

The Remotable Object section must include the **keyword = RemotingServer** entry. This entry indicates that the probe's Instrumenter should perform special processing for the point settings in this section. This special processing accomplishes two things. It instruments all methods on a class that inherits from System.MarshalByRefObject. Therefore, you need not specify which Remoting methods to instrument. All Remoting methods will be instrumented. This is also why there is no need for a method entry in this section. Second, this keyword isolates the instrumentation of methods that are implemented on a class that inherits from System.MarshalByRefObject to the specified class. This is important because there are many System classes and user classes that also inherit from System.MarshalByRefObject and you do not want to unintentionally instrument them.

Based on these observations, here is the recommended Remotable Object section:

```
[RemotableObject]
keyword = RemotingServer
class = HPSoftware.AM.Tests.Remoting.SimpleRemoting.MyRemotableObject
layer = RemotableObject
```

Now you can construct the optional Remoting Server section. You only need to create this section if you want to monitor the Server logic that is invoked independent of the Remoting methods.

When you run the Reflector against the SimpleConsoleServer.exe, you will see output that includes:

```
----------------------------------------------------------------------
Sample .points by Namespace
----------------------------------------------------------------------
[HPSoftware.AM.Tests.Remoting.SimpleRemoting]
class = !HPSoftware.AM.Tests.Remoting.SimpleRemoting.*
layer = HPSoftware/AM/Tests/Remoting/SimpleRemoting

----------------------------------------------------------------------
(1 classes) Namespace: HPSoftware.AM.Tests.Remoting.SimpleRemoting
----------------------------------------------------------------------
HPSoftware.AM.Tests.Remoting.SimpleRemoting.SimpleConsoleServer (7
Methods)
Equals System.Boolean(System.Object)
Finalize System.Void()
GetHashCode System.Int32()
GetType System.Type()
Main System.Void(System.String[])
MemberwiseClone System.Object()
ToString System.String()
```

As explained in "How to Configure and Set Up Points for Non-ASP.NET or Windows Applications" on page 74, you cannot just use the suggested points settings. You must ignore the Main() method.

Based on these observations, the following settings are the recommended settings for the SimpleConsoleServer.points file:

```
[RemotableObject]
keyword = RemotingServer
class = HPSoftware.AM.Tests.Remoting.SimpleRemoting.MyRemotableObject
layer = RemotableObject

[RemotingServer]
class = HPSoftware.AM.Tests.Remoting.SimpleRemoting.SimpleConsoleServer
ignoreMethod = Main
layer = RemotingServer
```

Finally, you can add other optional sections to instrument other methods that can be called by either the Remoting methods or the Remoting Server.

Understanding the Overhead of Custom Instrumentation

When creating custom instrumentation, beware of over-instrumenting the application because that can introduce excessive latency into the probed application. The custom instrumentation does not have the same impact on the method latency or the CPU overhead because the overhead of instrumentation is nearly fixed for every method because the amount of bytecode is almost always the same. The physical percentages of the CPU and latency overhead will vary in direct proportion to the length of time the method takes to execute.

For example, if a method takes 100ms and instrumentation makes it execute in 101ms, overhead is 1%. If a method takes 10ms and instrumentation changes its response to 11ms, overhead is 10%. If this method is not called very often, its overall latency effect on the application is minimal. However, the overall latency effect of an instrumented method that is called more frequently could have an impact on the latency of the application's response even though its overhead percentage is much smaller.

Unlike a traditional profiler that can profile every method called, HP Diagnostics uses bytecode instrumentation. This allows the default instrumentation to be selective so as to minimize the overhead caused by instrumentation to an average of 3-5%. Methods with higher latency overhead introduced by instrumentation are only instrumented when they are called infrequently in relation to other components in the application and when the instrumentation provides specific information needed for triage activities.

You should also consider Diagnostics data overhead when you are customizing the instrumentation for an application. The more methods you instrument, the more data the probe must serialize and pass over the network to the Diagnostics Server. You can tune the probe's default configuration so that it can adjust the volume of Diagnostics data to avoid any unnecessary effect on the performance of the system being monitored. Improper probe tuning can cause CPU, Memory, and Network overhead on the physical machine where your probe resides. For more information about managing Latency, CPU, Memory and Network overhead, see "Advanced .NET Agent Configuration" on page 170.

Default Layers for Typical .NET Applications

HP Diagnostics groups the performance metrics for classes and methods into *layers* and *sublayers* according to the instructions provided in the points file. The default layers were defined so that the performance metrics for processing in the application that used similar system resources could be reported together. The layers make it easier for you to isolate and identify the areas of the system that could be contributing to performance issues.

The following table lists the default layers and sublayers that are defined for typical .NET applications.

.NET Layers

Layer	Sublayers	Parent Layer
Web Tier	IIS	

Layer	Sublayers	Parent Layer
IIS	ExecutionSteps	
Database	ADO	
ADO	Execute Connection Fill Update Cache	Database
Messaging	Sender Receiver	
Web Services	Soap Http WCF	
LWMD		
HTTP Client		
Outbound Calls		

Chapter 6: Understanding the .NET Agent Configuration File

You control the configuration of the .NET Agent by modifying the elements and attributes in the .NET Agent configuration file: **<probe_install_dir>/etc/probe_config.xml**.

This chapter includes:

"Understanding .NET Agent Configuration File" below

".NET Agent Configuration Elements" on the next page

Understanding .NET Agent Configuration File

The topics in this section describe the elements and attributes that make up the .NET Agent configuration file **<probe_install_dir>/etc/probe_config.xml**.

Each element is defined by describing its purpose, attributes, and parent and children elements. For information on additional .NET Agent configuration elements specific to TransactionVision see the HP TransactionVision Deployment Guide.

.NET Agent Configuration Elements

<ali> element

Purpose

Enables ALI integration.

Attributes

Attributes	Valid Values	Default	Description
enabled	true false	false	Enable or disable the ALI integration. If enabled, build information (build number, build data and server) for a selected probe can be viewed in the Diagnostics Commander and in an HP AppPulse environment.

Elements

Number of Occurrences	zero or more
Parent Elements	probeconfig
Child Elements	none

Example

```
<ali enabled="false" />
```

<appdomain> element

Purpose

Builds an AppDomain inclusion list for processes that host multiple application domains. If no appdomain elements are defined for a process then all application domains for that process will be included.

Attributes

Attributes	Valid Values	Default	Description
enabled	true false	true	Determines if the AppDomain should be instrumented. Is overridden by enableallappdomains attribute of a process element. **Note:** When an AppDomain is enabled or disabled, you must restart the process for the change to take effect. For details on restarting IIS, see "Step 12. Restart IIS" on page 37. (To restart an application that is neither IIS hosted, nor running as a Windows Service, stop and start the application by whatever method is relevant for the application.)
name	string	none	Name of the .NET AppDomain. (IIS path qualified, see the example below.)
website	string	none	The name of the Website for those AppDomains that are Websites (information only)

Elements

Number of Occurrences	zero or more
Parent Elements	process
Child Elements	bufferpool, credentials, diagnosticsserver, mediator, id, ipaddress, logging, lwmd, modes, points, profiler, sample, trim, webserver, symbols, filter, topology

Example

```
<appdomain enabled="true" name="1/ROOT/MSPetShop"/>
Where 1/ROOT is the Website ID and MsPetShop is the Virtual DirName

<appdomain enabled="false" name="1/ROOT" website="Default Web Site">
  <points file="Default Web Site.points"/>
  <id probeid="Default Web Site" />
</appdomain>
```

<authentication> element

Purpose

List of authenticated user names and passwords.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
username	string	admin	User name account.	No
password	string	admin	Passwords must be generated using the passgen utility in the **<probe_install_dir>\bin** directory.	No

Elements

Number of Occurrences	zero to many
Parent Elements	profiler
Child Elements	none

Example

```
<profiler authenticate="true">
  <authentication username="Test" password="uU8X9zOtl6Twi7TkGAhQ="/>
</profiler>
```

<bufferpool> element

Purpose

Configures the bufferpool behavior.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
size	number	65536	Size of each buffer.	Yes
buffers	number	512	Number of buffers in pool.	Yes
sleep	number	1000	Number of milliseconds between flush checks.	Yes
expires	number	1000	Number of milliseconds before buffer expires.	Yes

Elements

Number of Occurrences	1 per parent
Parent Elements	appdomain, probeconfig, process
Child Elements	none

Example

```
<bufferpool size="65536" buffers="512" sleep="1000" expires="1000" />
```

<capturecookies> element

Purpose

Enables cookies capture for TV events.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
enabled	true false	true	Enables, disables capture of cookies for generated TV events	No

Elements

Number of Occurrences	1
Parent Elements	tv
Child Elements	none

Example

```
<tv eventthreads="3" eventthreadsleep="80" eventmemorythreshold="25000000"
configthreadsleep="10000" >
<timeskew historysize="24" checkinterval="300000" latencythreshold="100"
retrythreshold="8"/>
<transport type="sonicmq"
connectionstring="broker=myhost.mydomain.com;port=21111; user=;
password=;configurationqueue=TVISION.CONFIGURATION.QUEUE"/>
<capturecookies enabled=”true”/>
<capturehttpheaders enabled=””true/>
</tv>
```

<captureexceptions> element

Purpose

Enables and controls the stack trace capture for exceptions.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
enabled	true false	true	Enables exception capture.	No
capture_ args	true false	true	Enables (true) or disables (false) the display of method parameters in the Exception tab of a call profile.	No
max_per_ request	number	4	Maximum exceptions captured for one server request.	No
max_ stack_ size	number	0 (meaning no maximum)	Maximum size of the call stack for a captured exception.	No

Elements

Number of Occurrences	1
Parent Elements	probeconfig
Child Elements	include, exclude

Example

```
<captureexceptions enabled="true" max_per_request="4" max_stack_size="0"
capture_args="true">
```

<capturehttpheaders> element

Purpose

Enables http header capture for TV events.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
enabled	true false	true	Enables, disables capture of http headers for generated TV events	No

Elements

Number of Occurrences	1
Parent Elements	tv
Child Elements	none

Example

```
<tv eventthreads="3" eventthreadsleep="80" eventmemorythreshold="25000000"
configthreadsleep="10000" >
<timeskew historysize="24" checkinterval="300000" latencythreshold="100"
retrythreshold="8"/>
<transport type="sonicmq"
connectionstring="broker=myhost.mydomain.com;port=21111; user=;
password=;configurationqueue=TVISION.CONFIGURATION.QUEUE"/>
<capturecookies enabled=”true”/>
<capturehttpheaders enabled=””true/>
</tv>
```

<clientmonitoring> element

Purpose

This is the root element for configuring client monitoring for the .NET Agent.

Attributes

Attributes	Valid Values	Default	Description
enabled	true false	false	Enables/disables client monitoring
samplemethod	percent count period	percent	Specifies which method to use for sampling
samplerate	for **percent** rate must be 0-100 for **count** rate must be >1 for **period** rate must be one of standard Diagnostics time strings (3m for 3 minutes, 4s for 4 seconds, and so forth)	50	Specifies the rate for sampling

Elements

Number of Occurrences	1
Parent Elements	probeconfig
Child Elements	htmlinstrumentation, server, filter

Example

```
<clientmonitoring enabled="false" samplemethod="percent" samplerate="50" >
```

<consumeridrules> element

Purpose

This is the root element for configuring consumer ID rules.

Attributes

Attributes	Valid Values	Default	Description
enabled	true false	false	Enables consumer ID rule evaluation.

Elements

Number of Occurrences	1
Parent Elements	probeconfig
Child Elements	httpheaderules, iprules, soaprules

Example

```
<consumeridrules enabled="false">
```

<cputime> element

Purpose

Controls the **cputime** setting property.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
mode	none, serverrequest, method	serverrequest		No

Elements

Number of Occurrences	1
Parent Elements	probeconfig, process, or appdomain
Child Elements	none

Example

```
<cputime mode="serverrequest"/>
```

<credentials> element

Purpose

Supplies credentials that are used to validate for communication with the Diagnostics Server.

Attributes

Attributes	Valid Values	Default	Description
username	string	none	User name to validate with the Diagnostics Server.
password	string	none	Password to validate with the Diagnostics Server.
authenticate	true, false	true	Enables and disables authentication.

Elements

Number of Occurrences	1 per parent
Parent Elements	appdomain, probeconfig, process
Child Elements	none

Example

```
<credentials username="test" password="diag" authenticate="true"/>
```

<demomode> element

Purpose

This configures demo mode. Demo mode makes it easier to show capability and value of the .NET agent because it requires less custom points to be defined. With demomode turned on, all outbound calls will be shown irrespective of any other instrumentation.

Once the calls leading to the outbound calls of interest are identified then demomode should be turned off and "custom" instrumentation added to ensure that call stacks leading to the outbound calls are apparent.

It is recommended to TURN THIS OFF under production environments.

Demomode is used primarily to find outbound calls (webserver, http, remoteing, msmq) when the method making them is not instrumented. It is meant as a way to quickly find how applications may be connected without having to instrument application specific methods . This may be too noisy in production situations but is useful when you there is a lack of upstream instrumentation and you don't knowwhere the outbound call is being made from. It can be used for all kinds of applications including ASP.NET.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
enabled	true, false	false	Enables or disables demo mode.	No

Elements

Number of Occurrences	Zero or one.
Parent Elements	probeconfig
Child Elements	none

Example

```
<demomode enabled="false"/>
```

<depth> element

Purpose

Configures depth trimming.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
enabled	true false	true	Enables depth trimming.	No
depth	number	25	Sets the depth for depth trimming.	No

Elements

Number of Occurrences	1
Parent Elements	trim
Child Elements	none

Example

```
<trim>
  <depth enabled="true" depth="25"/>
</trim>
```

<diagnosticsserver> element

Purpose

Contains connection and settings information related to the Diagnostics Server which are used for enterprise mode.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
url	Registrar URL. http://<host>:<port>	none	URL to connect to registrar.	No
delay	number	2	Number of seconds to wait before registering.	Yes
keepalive	number	15	Number of seconds between keepalives.	No
proxy	URL of proxy	none	Registrar connection proxy.	
proxyuser	user id for proxy	none	Proxy user account.	
proxypassword	password for proxy	none	Proxy user account's password.	
registered_hostname	string	none	Name of host to register as (external name for firewall traversing).	Yes
register_byip	true, false	false	Register using ipaddress instead of hostname.	Yes
timeskewcheckinterval	number	60	Number of seconds to wait for getting the time skew from the Diagnostics server.	No

Elements

Number of Occurrences	1 per parent
Parent Elements	probeconfig
Child Elements	none

Example

This is a general example showing the setting for the <diagnosticsserver> element. The question marks (?) indicate that appropriate values need to be substituted.

```
<diagnosticsserver url="http://localhost:2006/commander" delay="2"
keepalive="15" proxy="?" proxyuser="?" proxypassword="?" registerhostname="?"
register_byip="false"/>
```

For the steps involved in using the registered_hostname attribute to override the default probe host machine name see "Overriding the Default Probe Host Machine Name" on page 192.

<exceptiontype> element

Purpose

Define an exception type.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
name	string	None	Class name of an exception.	No

Elements

Number of Occurrences	Zero to many
Parent Elements	include, exclude
Child Elements	None

Example

```
<exceptiontype name="System.DivideByZeroException"/>
```

<exclude> element (when parent is captureexceptions)

Purpose

Define a list of exceptions to exclude.

Attributes

None

Elements

Number of Occurrences	1
Parent Elements	captureexceptions
Child Elements	exceptiontype

Example

```
<exclude>
  <exceptiontype name="System.DivideByZeroException"/>
</exclude>
```

<exclude> element (when parent is lwmd)

Purpose

Define which collection classes to exclude from the Collections by Growth and Collections by Size tables in the .NET Profiler's Collections tab and the Diagnostics user interface's Collections view.

The specified collection classes may include classes that implement `ICollection`. Note that this setting does not affect the instrumentation of LWMD points; it only affects the presentation of the LWMD data and the amount of LWMD data that is sent to the Diagnostics Server.

Attributes

None

Elements

Number of Occurrences	Zero to many
Parent Elements	lwmd
Child Elements	None

Example

```
<lwmd enabled="true" sample="15s" autobaseline="1h" growth="10" size="10">
  <exclude>System.Collections.ArrayList</exclude>
  <exclude>System.Data.DataView</exclude>
</lwmd>
```

Note that System.Data.DataView implements System.Collections.ICollection.

<excludeassembly> element

Purpose

Excludes the instrumentation of an assembly. An assembly is an .exe or .dll file. Provides the ability to exclude sensitive assemblies from instrumentation (for example, when a product was used to obfuscate and encrypt code in sensitive assemblies and exceptions would be thrown if instrumented).

Add <excludeassembly name=<AssemblyNameToExclude> as a child to a process element.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
name	string	none	Name of assembly to exclude (without the file extension).	Yes

Elements

Number of Occurrences	zero to many
Parent Elements	process
Child Elements	none

Example

```
<process enablealldomains="true" name="ASP.NET">
  <logging level="" />
  <points file="ASP.NET.points" />
  <points file="ADO.points" />
  <points file="WCF.points" />

  <excludeassembly name="Acme.Encryption" />

  <appdomain enabled="false" name="TestWebService">
    <points file=" TestWebService .points" />
  </appdomain>
</process>
```

<excludesqlparam> element

Purpose

Excludes specific SQL Bind Parameters from being captured.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
name	string	none	Name of the SQL Bind Parameter to be excluded from capture for user needs (for example, for security reasons). This can be a list of Parameter Names.	No

Elements

Number of Occurrences	zero to many
Parent Elements	sqlparsing
Child Elements	none

Example

```
<sqlparsing mode="3" capturesqlparameters="true">
  <excludesqlparam  name="p__linq__1"/>
  <excludesqlparam  name="p__linq__0"/>
</sqlparsing>
```

<filter> element

Purpose

Filters out certain metrics that would skew the results or not be representative of the processing being monitored.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
firstserverrequest	true, false	false	Enables/disables skipping the collection of metrics for the first time a particular server requests (URL) gets run after application startup.	Yes

Elements

Number of Occurrences	1 per parent
Parent Elements	appdomain, probeconfig, process
Child Elements	none

Example

```
<filter firstserverrequest="false"/>
```

<filter> element

Purpose

Enables the inclusion or exclusion of web pages from client monitoring.

Attributes

Attributes	Valid Values	Default	Description
type	include exclude	exclude	Specifies whether to include or exclude web pages from client monitoring

Elements

Number of Occurrences	1
Parent Elements	clientmonitoring
Child Elements	url

Example

```
<filter type="include">
     <url name=".*\.aspx" />
</filter>
```

<gentvhttpeventforwcf> element

Purpose

Setting this option enables generation of a TransactionVision event for a WCF service with any binding that uses IIS (http based) hosting. Some WCF services may use a custom or private binding that is not supported as a true web service and in these types of cases TransactionVision web service events would not be generated unless you enable this option.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
enabled	true, false	false	Enables/disables the generation of an http event for a WCF service with any binding that uses IIS (http based) hosting. If enabled, provides TransactionVision web service events.	No

Elements

Number of Occurrences	Zero to one.
Parent Elements	probeconfig, process, appdomain
Child Elements	none

Example

```
<gentvhttpeventforwcf enabled="true"/>
```

<htmlinstrumentation> element

Purpose

Enables configuring an alternate instrumentation file to be used for client monitoring. The file must be located in the /etc directory.

Note: If an htmlinstrumentation file is set, server element settings are ignored.

Attributes

Attributes	Valid Values	Default	Description
File	HPRUMCMInst.hpcm	null	The name of the file containing alternate (RUM) client monitoring instrumentation. The file must be located in the etc folder.

Elements

Number of Occurrences	1
Parent Elements	clientmonitoring
Child Elements	none

Example

```
<htmlinstrumentation file="HPDefaultInst.hpcm" />
```

<httpcaptureparams> element

Purpose

Specifies how to configure and capture selected query parameters of HTTP Requests by .NET web applications.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
enabled	true, false	false	Enables/disables HTTP parameter capture.	No
capturequerystring	true, false	false	Enables/disables the query string capture. The query string is captured as a Server Request instance property. This attribute works independently of the enabled attribute which is used to control the parameter capture list.	No
param name	"Genre" for example	none	Specifies which query parameter by name should be captured as part of the Server Request Name.	No

Number of Occurrences	Zero to one.
Parent Elements	probeconfig, process
Child Elements	param

Example

For the HTTP URL `http://MachineName/MVC3/MusicStore/Store/Browse?Genre=Rock&Artist=Punk` with this configuration in the probe_config.xml file:

```
<httpcaptureparams enabled="true" capturequerystring="true" >
   <param name="Genre"/>
   <param name="accounttype"/>
</httpcaptureparams>
```

You see the following server requests:

```
/MVC3/MusicStore/Store/Browse?Genre=Alternative
/MVC3/MusicStore/Store/Browse?Genre=Blues
/MVC3/MusicStore/Store/Browse?Genre=Classical
/MVC3/MusicStore/Store/Browse?Genre=Disco
```

```
/MVC3/MusicStore/Store/Browse?Genre=Latin
/MVC3/MusicStore/Store/Browse?Genre=Metal
/MVC3/MusicStore/Store/Browse?Genre=Pop
/MVC3/MusicStore/Store/Browse?Genre=Reggae
/MVC3/MusicStore/Store/Browse?Genre=Rock
```

<httpclient> element

Purpose

This configures whether the URL will be included as part of an HTTP outbound call's identity. The default is true and should be kept so unless there are many distinct URLs for the outbound HTTP calls. This could potentially overwhelm the performance of the Diagnostics Server because of the number outbound calls created (one for each distinct URL). You may also want to turn it off if you do not care about the URL of the HTTP outbound call. The identity of the HTTP outbound call will then be the Server and port number to which the request is being made to.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
showurl	true, false	true	Enables/disables the inclusion of the URL as part of the identity of an outbound call made by a client using HTTP. Setting to false can be used to protect against symbol table explosion on the server/agent side if there are too many distinct http client calls. The value should be set to false for REST service client applications	No

Elements

Number of Occurrences	Zero to one.
Parent Elements	probeconfig, process, appdomain
Child Elements	none

Example

```
<httpclient showurl="true"/>
```

<httpheaderrule> element

Purpose

Defines a consumer ID rule for HTTP headers.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
id	string	None	ID of the rule.	No
rule	string	None	A regular expression that is used to match against the URL that the HTTP request is being sent to by the consumer.	No
consumeridfield	string	None	Name of the header to use as the consumer ID.	No

Elements

Number of Occurrences	Zero to many
Parent Elements	httpheaderrules
Child Elements	None

Example

```
<httpheaderrule id="httpHeader 1" rule="/Webservice/.*"
consumeridfield="Caller"/>
```

<httpheaderrules> element

Purpose

This element contains all of the <httpheaderrule> elements.

Attributes

None

Elements

Number of Occurrences	1
Parent Elements	consmeridrule
Child Elements	httpheaderule

Example

```
<httpheaderrules>
</httpheaderrules>
```

<id> element

Purpose

Provides probe id and probe group id.

Attributes

Attribute	Valid Values	Default	Description
probeid	String containing: Letters, digits, underscore, dash, period and internally defined $() variable values: $(APPDOMAIN), $(MACHINENAME), $(WEBSITENAME), $(PID)	$(MACHINENAME)_ $(APPDOMAIN).NET	The name of the probe as recognized by LoadRunner / Performance Center and System Health.
probegroup	string	Default	Defines the grouping recognized by the Diagnostics Server for reporting of system metrics and probe metrics.

Elements

Number of Occurrences	1 per parent
Parent Elements	probeconfig, process, appdomain
Child Elements	none

Example

Default setting example.

```
<id probeid="$(MACHINENAME)_$(APPDOMAIN).NET" probegroup="Default"/>
```

Example

Example for a probe running in a LoadRunner 8.1 environment reporting to "myDiagServer" with the probe's name comprised of valid characters, the name of the Web site the application is deployed under, plus the name of the machine the application is deployed on.

```
<id probeid="LR_81_$(WEBSITENAME)_$(MACHINENAME).NET" probegroup="LR_81_
myDiagServer"/>
```

<include> element (when parent is captureexceptions)

Purpose

Define a list of exceptions to include.

Attributes

None

Elements

Number of Occurrences	1
Parent Elements	captureexceptions
Child Elements	exceptiontype

Example

```
<include>
  <exceptiontype name="System.DivideByZeroException"/>
</include>
```

<include> element (when parent is lwmd)

Purpose

Define which collections to include to the exclusion of others.

Attributes

None

Elements

Number of Occurrences	Zero to many
Parent Elements	lwmd
Child Elements	None

Example

```
<include>System.Collections.Hashtable</include>
<include>System.Collections.ArrayList</include>
```

<instrumentation> element

Purpose

Contains logging configuration for instrumenter.

Attributes

None.

Elements

Number of Occurrences	1 per parent
Parent Elements	probeconfig, process
Child Elements	logging

Example

```
<instrumentation>
  <logging level="property lwmd" />
</instrumentation>
```

<iprule> element

Purpose

Defines a consumer ID rule for IP addresses.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
id	string	None	Enables consumer ID rule evaluation.	No
rule	string	None	Define an IP address, or a range of addresses, to be assigned to a consumer ID.	No
consumerid	string	None	The consumer ID to use if there is a match on the rule.	

Elements

Number of Occurrences	zero to many
Parent Elements	iprules
Child Elements	none

Example

```
<iprule id="IpTest1" rule="43.*.1-20.*" consumerid="HP"/>
```

<iprules> element

Purpose

This element contains all of the <iprule> elements.

Attributes

None

Elements

Number of Occurrences	1
Parent Elements	consumeridrules
Child Elements	iprule

Example

```
<iprules>
</iprules>
```

<latency> element

Purpose

Configures latency trimming.

Attributes

Attributes	Valid Values	Default	Description
enabled	true false	true	Enables latency trimming.
throttle	true false	true	Enables latency trimming throttling.
min	number	2	Minimum latency threshold.
max	number	100	Maximum latency threshold.
increment	number	2	Threshold increment.
increment threshold	number	75	The percentage of the buffer usage before the throttling should be incremented.
decrement threshold	number	50	The percentage of the buffer usage before the throttling should be decremented.

Elements

Number of Occurrences	1
Parent Elements	trim
Child Elements	none

Example

```
<trim>
  <latency enabled="true" throttle="true" min="2" max="100" increment="2"
incrementthreshold="75" decrementthreshold="50"/>
</trim>
```

<logdirmgr> element

Purpose

Contains the configuration for the log directory manager. The logdirmgr monitors the log directory to ensure that it does not grow unbounded. The logdirmgr scans the logs periodically as indicated by the scaninterval. If the size has exceeded the size indicated by maxdirsize the logdirmgr deletes the oldest files until the size no longer is greater than the maxdirsize.

Important: The account under which the .NET process is running (for IIS the AppPool Account) has to be provided **delete** privileges on the log folder. This is not available by default on the NETWORK SRERVICE account or the App Pool Identity Account (which is the default Application Pool Account).

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
enabled	true false	true		
maxdirsize	number	1024	Largest size in MB to which the log directory can grow. Must be at least 1(MB).	No
scaninterval	number	30	How often in minutes that the manager scans the logs to check for growth and size. Must be at least 10 (minutes).	No

Elements

Number of Occurrences	1 per parent
Parent Elements	probeconfig
Child Elements	none

Example

```
<logdirmgr enabled="true" maxdirsize="1" scaninterval="10"/>
```

<logging> element (when parent is instrumentation)

Purpose

Sets the logging level for the .NET Agent instrumentation processing.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
level	off assert break severe warning info _____ debug points eh sig chi cil classmap ilasm symbols deepmode load all checksum property remoting lwmd http	"" which is equivalent to "info"	Level of logging.	No
threadids	true false	true	Should thread IDs be included in the log.	

Valid values below "info" should typically not be used. These are diagnostic settings that can produce extremely large log files.

Elements

Number of Occurrences	zero to many

Parent Elements	instrumentation
Child Elements	none

Example

```
<instrumentation>
  <logging level="warning" />
</instrumentation>
```

<logging> element (when parent is appdomain, probeconfig, or process)

Purpose

Sets the logging level for the .NET Agent processing for monitoring and reporting application performance.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
level	off severe warning info _____ debug events property webserver http symbols probemetrics registrar threadpool authentication bufferpool rum bacforsoa vmware exceptions| tvdebug	"" which is equivalent to "info"		No
max	number	10	The maximum size of a probe log file. After the log reaches this size no more logging will occur.	No

Valid values below "info" should typically not be used. These are Diagnostic settings that can produce extremely large log files.

Elements

Number of Occurrences	

Parent Elements	appdomain, probeconfig, process
Child Elements	none

Example

```
<logging max="10" level="INFO"/>
```

<lwmd> element

Purpose

Configures the Light-Weight Memory Diagnostics (LWMD) feature.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
enabled	true false	false	Enables sampling for lwmd capturing.	No
sample	string	1m	Sample interval (h-hour/m-minute/ s-second).	
autobaseline	string	1h	Auto baseline interval.	
manualbaseline	string	none	Manual baseline time.	
growth	number	15	Number of collections to growth track.	
size	number	15	Number of collections to size track.	

Elements

Number of Occurrences	1 per parent
Parent Elements	appdomain, probeconfig, process
Child Elements	exclude, include

Example

```
<lwmd enabled="false" sample="1m" autobaseline="1h" manualbaseline= "?"
growth="15" size="15"/>
```

<mediator> element

Purpose

Specifies the diagnostics server that is in the Mediator mode to which events are to be sent when in the enterprise mode.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
host	host name	none	Name of mediator.	No
port	number	2612	Mediator port.	No
ssl	true/false	false	When the Diagnostics Server URL starts with `http` the default is `false`. When the Diagnostics URL starts with `https` the default is `true`.	Yes
metrichost	string		The host to which metric data is sent.	No
metricport	number	2006	The port to which the probe sends the probe metrics such as heap usage and availability.	No
block	true/false	false	Block until mediator connection established.	
ipaddress			local ipaddress to use when connecting to the eventserver.	
localportstart	number	4000	Beginning of port range to use for tcp event channel connection to the Diagnostics Server in Mediator mode. Used only when ipaddress is specified.	
localportend	number	5000	End of port range to use for tcp event channel connection to the Diagnostics Server in Mediator mode. Used only when ipaddress is specified.	

Elements

Number of Occurrences	1 per parent
Parent Elements	appdomain, probeconfig, process
Child Elements	none

Example

```
<mediator host="localhost" port="2612" ssl="false" metricport="2006"
block="false" ipaddress="16.255.18.99" localportstart="4000"
localportend="5000"/>
```

<metrics> element

Purpose

This element contains all of the <metric> elements.

Attributes

None

Elements

Number of Occurrences	1 per parent
Parent Elements	appdomain, process
Child Elements	metric

Example

```
<metrics>
  <metric name="% Time in GC" group="Memory" units="percent" category=".NET CLR
Memory" counter="% Time in GC"/>
</metrics>
```

<metric> element

Purpose

Specifies additional probe metrics that you want the Diagnostics .NET to collect from perfmon. See "Collecting Additional Probe Metrics or Modifying Probe Metrics" on page 199 for additional information.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
name	string		Name of the metric as you would like to see it in the Diagnostics UI.	Yes
group	string		Group (Category) of the metric as you would like to see it in the Diagnostics UI.	Yes
units	microseconds, milliseconds, seconds, minutes, hours, days, bytes, kilobytes, megabytes, gigabytes, count, percent, fraction_percent, load, status		Units of measure for the perfmon metric.	Yes
category	string		The performance counter category as specified in perfmon.	Yes
counter	string		The performance counter as specified in perfmon	Yes

Note: The instance of the counter is automatically assigned as the process instance for the counter or application domain instance for ASP.NET application counters. Counters that do not have process or application domain instances are not collected; you should define system metrics instead.

Elements

Number of Occurrences	1 or more per parent
Parent Elements	metrics
Child Elements	none

Example

```
<metrics>
  <metric name="% Time in GC" group="Memory" units="percent" category=".NET CLR
Memory" counter="% Time in GC"/>
</metrics>
```

<modes> element

Purpose

Specifies which product mode(s) the .NET Agent should run in. See "Controlling Which HP Software Products the Agent can Work With" on page 177 for more information about using the different modes.

The <modes> element is also used in determining usage against the HP Diagnostics license capacity.

See the chapter "Licensing HP Diagnostics" in the HP Diagnostics Server Installation and Administration Guide for more information.

The value of the <modes> element is initially set at the time you install the agent.

The .NET agent can set in different modes to do the following:

- Monitor applications from development through pre-production testing and into production.
- Used with other HP Software products.
- Used as a standalone Diagnostics Java Profiler not reporting to a server or to other HP Software products.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
enterprise	true false	Depends on mode chosen in installation. true if pro is false false if pro is true	Sets agent to run in enterprise mode (probe is working with Diagnostics Server). Enterprise mode is like a combination of *ad*, *am* and *pro* mode. It will capture data for LoadRunner runs as well as data outside of LoadRunner runs. Enterprise mode is the default for .NET Agents (if you don't specify AD or AM mode). In Enterprise mode the agents are counted against the AM license capacity.	No
ent	true false	Depends on mode chosen in installation. true if pro is false false if pro is true	This is a short form of the enterprise attribute.	No

Attributes	Valid Values	Default	Description	Requires Application Restart
ad	true false	false	*ad* mode supersedes all other modes. If *ad* mode and any other modes are set, then mode will be set to *ad*. In *ad* mode the .NET Agent will only capture runs from LoadRunner and put the results in a specific database for that run (for example, Default21). Agents in AD mode will only be counted against AD license capacity when the probe is running in a LoadRunner or Performance Center test run. When not in a test run the agent does not count against license capacity. For example if 20 probes are installed in LoadRunner/Performance Center AD mode but only 5 are in a run, then only 5 are counted against AD license capacity.	No
am	true false	false	*am* mode supersedes all other modes except for *ad*. In *am* mode the .NET agent will ignore runs. If LoadRunner is executing an application then you will see the data in the normal Diagnostics database. Agents in AM mode will always be counted against the AM license capacity.	No
pro	true false	Depends on mode chosen in installation. true if enterprise is false false if enterprise is true	Sets the agent to run in Profiler mode. This mode sends data to the profiler. This mode can be combined with other modes. Agents in pro mode are not counted against license capacity.	No
tv	true false	false	Enables the capture of TransactionVision events. See "About Configuration of the .NET Agent for TransactionVision" on page 39 for details on setting transport and other TV options. This mode will send events to TransactionVision. This mode can be combined with other modes and in tv mode agents are not counted against Diagnostics's license capacity	Yes

Elements

Number of Occurrences	1 per parent
Parent Elements	appdomain, probeconfig, process
Child Elements	none

Example

```
<modes enterprise="false" ad="false" am="false" pro="true"/>
```

<param> element

Purpose

Specifies a query parameter to capture in an HTTP request.

Attributes

Attributes	Valid Values	Default	Description
name	string	none	Name of the .NET process that these setting apply to.

None.

Number of Occurrences	Zero to many.
Parent Elements	httpcaptureparams
Child Elements	none

Example

For the HTTP URL `http://MachineName/MVC3/MusicStore/Store/Browse?Genre=Rock&Artist=Punk` with this configuration in the probe_config.xml file:

```
<httpcaptureparams enabled="true" capturequerystring="true" >
   <param name="Genre"/>
   <param name="accounttype"/>
</httpcaptureparams>
```

<points> element

Purpose

Specifies the capture points file to use for instrumentation.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
file	string	none	Name of instrumentation capture points file.	Yes

Elements

Number of Occurrences	zero or more
Parent Elements	appdomain, process
Child Elements	none

Example

```
<points file="ASP.NET.points"/>
```

<probeconfig> element

Purpose

Provides single containing root element for the .NET Agent configuration.

Attributes

None.

Elements

Number of Occurrences	1
Parent Elements	None
Child Elements	appdomain, bufferpool, captureexceptions, consumeridrules, credentials, diagnosticsserver, eventserverhost, id, instrumentation, ipaddress, logging, lwmd, mediator, modes, points, process, profiler, rum, sample, soappayload, trim, webserver, topology, vmware, xvm

Example

```
<probeconfig>
</probeconfig>
```

<process> element

Purpose

Provides an inclusion filter list of which processes will be monitored.

If no process elements are defined then no processes will be monitored.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
enablealldomains	true false	true	When set to true the enable attribute on all AppDomains that are part of the process is overriden so that all will be enabled.	Yes
name	string	none	Name of the .NET process that these setting apply to.	Yes

These are the rules for the enablealldomains attribute of the <process> element:

- enablealldomains = false : If there are no domains in the list of <appdomain> then no AppDomains should be enabled.
- enablealldomains = false : If there are domains in the list of <appdomain> then AppDomains should be enabled if the "enable" attribute is set to true or not defined in the enable attribute of the <appdomain>.
- enablealldomains = true : If there are domains in the list of <appdomain> then only AppDomains in the list should be enabled disregarding their "enable" attribute.
- enablealldomains = true : If there are no domains in the list of <appdomain> then all AppDomains should be enabled.
- enablealldomains attribute is not defined: same as if enablealldomains = true.

Elements

Number of Occurrences	zero or more
Parent Elements	probeconfig
Child Elements	appdomain, bufferpool, credentials, diagnosticsserver, mediator, id, instrumentation, ipaddress, logging, lwmd, modes, points, profiler, sample, trim, webserver, filter, symbols, topology

Example

```
<process enablealldomains="true" name="ASP.NET">
```

<profiler> element

Purpose

Contains settings for the Profiler feature.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
authenticate	true, false	none	Enables/Disables authentication of incoming Profiler connection requests. Changes to this attribute setting are applied dynamically; you do not need to restart the application or the probe.	No
register	true, false	false	Tells the probe to register even if it is in Profiler only mode.	No
samples	number	60	Tells the Profiler how many samples to keep for lwmd/heap trending.	No
best	number	1	The number of fastest instance trees to keeps.	No
worst	number	3	The number of slowest instance trees to keep.	No
inactivitytimeout	string	10m	The length of time that the Profiler continues to run after the user has stopped interacting with the Profiler.	No
disableremoteaccess	true, false	false	Disables remote access to the Profiler, thus not exposing the User/Password, and still be able to telnet/RemoteDeskTop into the machine and run the Profiler locally.	No

Elements

Number of Occurrences	1 per parent
Parent Elements	appdomain, probeconfig, process
Child Elements	authentication

Example

```
<profiler authenticate="true" register="false" samples="60" best="1" worst="3"
inactivitytimeout="10m">
  <authentication username="admin" password="admin"/>
</profiler>
```

<rum> element

Purpose

Controls the settings for Real User Monitoring.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
enable	true false	true	Enables or disables the RUM Integration feature.	No
responseheader	string	X-HP-CAM-COLOR	The name of the http header whose value contains the Diagnostics to RUM integration information.	No
encryptedkey	string		The encrypted key must be generated using the passgen utility in the <probe_install_dir>\bin directory.	No

Elements

Number of Occurrences	1 per parent
Parent Elements	probeconfig
Child Elements	none

Example

```
<rum enabled="true" responseheader="X-HP-CAM-COLOR"
encryptedkey="OBF:3pe941vx43903wre40303xxz3q6r42ob43n93wre3io03xjs40h940pc3wir3q
233jur3zir3yi03zir3vc03wre3xpi3r8o3olr44na3zor3v6m3vc03zir44u03ohb3rdi3xjs3wx03v
6m3zor3yc63zor3jqz3q6r3wd740vi40b53xpi3ike3wx043gp42ur3q233y3r3zwy3wx0432i42293p
9p"/>
```

To create the encrypted key, use the PassGen utility as follows:

```
cd <installdir>/bin
PassGen /system encryptionKey
```

Where **encryptionKey** is a string of alpha-numeric characters with a maximum length of 128 characters. The encryptedkey is shown on stdout.

passgen example:

```
PassGen /system TheLazyFoxJumpedHigh
```

Returns:

```
OBF:3q6r3xxz3y3r3xjs3wx03yc63n0r3lbr3vc03wd745893wre44u0413j3kn93zwy40vi432i44fr
3m453m894493439040pc40303kjd419r44na3wx0451h3wir3v6m3lfr3mwj3yi03wre3xpi3xxz3y3r
3q23
```

<sample> element

Purpose

Sets the sampling type and rate.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
method	percent, count, period	percent	Sets the sampling method: for percent rate must be 0-100 for count rate must be >1 for period rate must be one of standard Diagnostics time strings (3m for 3 minutes, 4s for 4 seconds, and so forth)	No
rate	number	0	Sets the sampling rate for percent type.	No

Elements

Number of Occurrences	1 per parent
Parent Elements	appdomain, probeconfig, process, ws
Child Elements	none

Example

```
<xvm><ws><sample method="percent" rate="50"/></ ws ></xvm>
Sampling is a random percentage rate.

<xvm>< ws ><sample method="count" rate="50"/></ ws ></xvm>
Sampling is once every rate count.

<xvm>< ws ><sample method="period" rate="60000"/></ ws ></xvm>
```

<server> element

Purpose

Configures the scripts and URLs to load for client monitoring instrumentation.

Note: If an htmlinstrumentation file is set, server element settings are ignored.

Attributes

Attributes	Valid Values	Default	Description
scriptsurl	Example: http://Mediatorhost/ClientMon/boomerang-min.js	http://Mediatorhost:port/boomerang-min.js	Defines the script and the URL to load for instrumentation
beaconurl	Example: http://Mediatorhost/ClientMonitoring/B	http://Mediatorhost:port/ClientMonitoring/B	Defines the script and the URL to load for instrumentation

Elements

Number of Occurrences	1
Parent Elements	clientmonitoring
Child Elements	none

Example

```
<server scriptsurl="boomerang-min.js" beaconurl="ClientMonitoring/B" />
```

<soapcapture> element

Purpose

Configures whether SOAP requests and responses are captured.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
enabled	true false	true	Enables or disables the capture of SOAP requests and responses. If this is disabled it will affect the following: SOAP request capture for SOAP faults SOAP requests and responses capture for TV mode ConsumerID assigned via the SOAP rules.	No
maxsize	number	0	This is an optional attribute that specifies the maximum size in characters of the SOAP request or response captured. 0 indicates unlimited.	No

Elements

Number of Occurrences	one per parent
Parent Elements	probeconfig
Child Elements	none

Example

```
<soapcapture enabled="true" maxsize="0" />
```

<soaprequestforsoapfault> element

Purpose

Configures SOAP request capture (including payloads) on SOAP Faults. Payloads can contain sensitive information such as credit card numbers so this element is disabled by default.

NOTE: If the <soapcapture> element is disabled it will override the <soaprequestforsoapfault> setting. Please refer to the documentation for the <soapcapture> element.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
enabled	true false	false	Enables or disables the SOAP request capture on SOAP fault feature. Disabled by default.	No
maxsize	number	5000	This is an optional attribute that specifies the maximum size in characters of SOAP request capture. If not present the Default value is used. If present and an error is made in the setting, the Default value is used.	No

Elements

Number of Occurrences	one per parent
Parent Elements	probeconfig
Child Elements	none

Example

```
<soaprequestforsoapfault enabled="true" maxsize="5000" />
```

<soaprule> element

Purpose

Defines a consumer ID rule for SOAP headers.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
id	string	None	ID of the rule.	No
rule	string	None	A regular expression that is used to match against the web service name being called by the consumer.	No
consumeridfield	string	None	The element in the SOAP header to get the value for to use as the consumer ID.	No
location	soap-header, soap-body, soap-envelope, Not set	Not set	The location within the SOAP payload where the soaprule applies.	No

Elements

Number of Occurrences	zero to many
Parent Elements	soaprules
Child Elements	none

Example

```
<soaprule id="SOAP1" rule="TestService2" consumeridfield="Caller"/>
```

<soaprules> element

Purpose

This element contains all of the <soaprule> elements.

Attributes

None.

Elements

Number of Occurrences	1
Parent Elements	consumeridrules
Child Elements	soaprules

Example

```
<soaprules>
</soaprules>
```

<sqlparsing> element

Purpose

This element is used to indicate in what mode SQL queries should be parsed. If there are a large number of SQL queries using literals it can overwhelm the server symbol table so the default is set to mode 3 to avoid this problem.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
mode	1, 2, 3, 4	3	Boolean property to turn asynchronous stack trace sampling on or off.	No
keywordsfile	string	None	Optionally allows you to specify a file containing keywords you want the agent to find in the SQL statement and highlight in upppercase when stored or displayed by Diagnostics. This helps ensure similar queries are recognized as the same query irrespective of case.	Yes
capturesqlparameters	true false	false	Turns capturing of SQL Bind Parameters for the Diagnostics Agent on or off.	No

Elements

Number of Occurrences	1
Parent Elements	probeconfig
Child Elements	excludesqlparam

Example

```
<sqlparsing mode="4" capturesqlparameters="false"
keywordsfile="C:\myfolder\mykeyword.txt"/>
```

<stacktracesampling> element

Purpose

Enables/disables and configures asynchronous thread stack trace sampling.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
enabled	true false	true	Enables or disables asynchronous stack trace sampling feature.	No
tardymethodlatency	number greater than 20	150	Minimum time (in millisecs) that an instrumented method must run without hitting any instrumentation points before stack trace sampling is attempted for this method. The purpose of this property is to control the overhead of sampling by limiting the stack trace collection to the most critical cases.	No
rate	number greater than 20	100	The time (in millisecs) that must elapse before the next consecutive sampling attempt is made. Small values cause frequent sampling, thus providing rich data, but at the cost of increased overhead. Large values cause many methods to miss most of the samples, thus required you to hunt for additional details in multiple saved instances, which may not be there. The overhead caused by frequent sampling affects primarily the latency of server requests. The overall CPU usage by the probe may go up as well, but this effect is not as profound as the latency increase. For machines with many CPUs, the process CPU consumption may actually go down (and it is not a good thing).	No
outboundcalls	true false	false	Turn asynchronous stack trace sampling on or off for outbound calls/	No

Elements

Number of Occurrences	1
Parent Elements	probeconfig
Child Elements	

Example

```
<stacktracesampling  enabled="true" tardymethodlatency="150" rate="100"
outboundcalls="false"/>
```

This statement enables stack trace sampling with the shown configuration.

<symbols> element

Purpose

Limits the number of unique URIs and SQL strings that can be captured to control the amount of memory consumed.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
maxuri	number	1000	Sets the top limit for number of unique URIs that can be captured.	No
maxuriname	string	Maximum number of unique URIs exceeded		No
maxsql	number	1000	Sets the top limit for number of unique URIs that can be captured.	No
maxsqlname	string	Maximum number of unique SQLs exceeded		No
usehttpmethod	true false	true	**true.** Use the HTTP method (such as PUT, GET, POST, and so forth) as the root method (identifying component) for each HTTP/S Server Request. This generates a separate HTTP Server Request for each HTTP method to the same URL. **false.** The root method (identifying component) for an HTTP Server Request is 'Server.Request'. This generates one HTTP Server Request for all HTTP methods to the same URL.	No

Elements

Number of Occurrences	1 per parent
Parent Elements	appdomain, probeconfig, process
Child Elements	urireplacepattern

Example

```
<symbols maxuri="1000" maxuriname="Maximum number of unique URIs exceeded"
maxsql="1000" maxsqlname="Maximum number of unique SQLs exceeded"
usehttpmethod="true"/>
```

<timeskew> element

Purpose

Used in configuring HP TransactionVision. Calculates the time difference between the time server and the host on which the .NET Agent is running. The frequency of checking with the time server can be configured.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
historysize	number	24	(Read on startup) number of time skew samples to store and compare for best sample.	Yes
checkinterval	number	300,000 ms.	(Dynamic) The time in milliseconds to wait before checking the time server for the skew time calculation.	No
latencythreshold	number	100 ms.	(Dynamic) The maximum time in milliseconds a reply from a time server can take for a valid time skew value.	Yes
retrythreshold	number	8	(Dynamic) Number of times to try when request to time server fails.	Yes

Elements

Number of Occurrences	1 (one)
Parent Elements	tv
Child Elements	none

Example

```
<timeskew historysize="24" checkinterval="300000" latencythreshold="100"
retrythreshold="8"/>
```

<topology> element

Purpose

Controls whether topology information will be collected and sent to the Diagnostics server.

Attributes

Attributes	Valid Values	Default	Description
enable	true false	true	Enables gathering topology information and passing it to the Diagnostics Server.

Elements

Number of Occurrences	1
Parent Elements	<probeconfig>, <process>, or <appdomain>
Child Elements	none

Example

```
<topology enable="true">
```

<transport> element

Purpose

Configure the events channel used by TransactionVision.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
type	mqseries sonicmq	sonicmq	The event transport provider being used by the Agent.	Yes
connectionString	See below.		The connection information for the event transport provider.	Yes

conectionString Syntax when type=sonicmq

```
broker = <broker>; port = <port>; user = <user>; password =<password>;
configurationQueue = <configurationQueue>
```

Where:	Is:
broker	Host name on which the Sonic broker is running. Typically this will be the Analyzer hostname.
port	The port on which the broker communicates. By default, 21111.
user	User id if required by SonicMQ installation for connection. By default, no username is required.
password	Password if required by SonicMQ installation for connection. This is in the obfuscated form created by using the PassGen utility. By default, no password is required. For more information about **PassGen**, see "Administration Utilities" in the *BSM Application Administration User Guide.*
configurationQueue	Name of the queue which has the configuration messages for the .NET TransactionVision Agent.

conectionString Syntax when type=mqseries

```
host= <host>; queuemanager=<queuemanager>; port= <port>; channel=,channel>
configurationQueue = <configurationQueue>
```

Where:	Is:
host	Host on which the TransactionVision configuration queue is hosted.

queuemanager	Name of the queuemanager.
port	MQSeries port on which the QueueManager communicates.
channel	MQSeries channel which will be used to communicate.
configurationQueue	Name of the queue which has the configuration messages for the .NET TransactionVision Agent.

Elements

Number of Occurrences	1 (one)
Parent Elements	tv
Child Elements	None

Example

For SonicMQ:

```
<transport type="sonicmq" connectionstring="broker=brokerHost;
  port=21111; user=; password=;
  configurationqueue=TVISION.CONFIGURATION.QUEUE"/>
```

For MQ Series:

```
<transport type="mqseries" connectionstring="host=mqHost;
  queuemanager=; port=1414; channel=TRADING.CHL;
  configurationqueue=TVISION.CONFIGURATION.QUEUE"/>
```

<trim> element

Purpose

Configures the trimming feature to reduce data volume transferred between the probe and the Diagnostics Server.

The Profiler user interface ignores all configured trim settings, for example, depth trimming and latency trimming, as the Profiler does not require that any data be sent to the Diagnostics Server.

Attributes

None.

Elements

Number of Occurrences	1 per parent
Parent Elements	appdomain, probeconfig, process
Child Elements	depth, latency

Example

```
<trim>
</trim>
```

<tv> element

Purpose

Configure the .NET Agent for use with TransactionVision.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
eventthreads	number	3	(Read on startup) The number of threads spawned by the Agent to send events to the Analyzer.	Yes
eventthreadsleep	number	100	(Dynamic) The time in milliseconds the event thread sleeps after sending a message(event package).	No
eventmemorythreshold	number	25,000,000	(Dynamic) The memory consumed by the internal buffer (Q) after which the Agent will try and send the message on the application thread.	No
configthreadsleep	number	10,000	(Dynamic) The time in milliseconds the event thread sleeps after browsing the configuration queue.	No

Elements

Number of Occurrences	1 (one)
Parent Elements	ProbeConfig
Child Elements	transport, timeskew

Example

```
<tv eventthreads="3" eventthreadsleep="80"
eventmemorythreshold="25000000" configthreadsleep="10000" >
  <timeskew historysize="24" checkinterval="300000" latencythreshold="100"
    retrythreshold="8"/>
    <transport type="sonicmq"
    connectionstring="broker=myhost.mydomain.com;
    port=21111; user=; password=;
    configurationqueue=TVISION.CONFIGURATION.QUEUE"/>
</tv>
```

<uriautocollapsing> element

Purpose

Configures automatic URI collapsing–the detection and trimming of server requests to avoid flooding the server symbol table with a large number of unique server requests.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
enabled	true false	true	Enables automatic URI collapsing.	No
limits	numbers separated by "/"	120/60/25/10	The maximum number of path segments allowed for each segment position, provided all of the preceding path segments are equal. The last specified value extends for all unspecified segments, that is, specifications 80/90/20 and 80/90/20/20/20 are equivalent.	No

Elements

Number of Occurrences	1
Parent Elements	probeconfig, symbols
Child Elements	none

Example

```
<symbols>
  <uriautocollapsing enabled="true" limits="120/60/25/10"/>
</symbols>
```

Once the limit for the fourth path segment is exceeded, URIs of that form are collapsed. For instance, assume the application receives the following URIs:

/a/b/c/01
/a/b/c/02
...
/a/b/c/11

Because the limit for the fourth path segment is exceeded, all future incoming URIs of that form will be replaced by /a/b/c/*.

The following screen shots show before and after automatic URI collapsing. The third segment of the URI path exceeds the specified limit so it is collapsed.

Status	Chart	Server Request	Probe	Latency % Over Thr...	Latency	CPU (Avg)
		/CallChain/CallChainForm.aspx/a19/b3	ROS8423...		1.7 ms	0.0 µs
		/CallChain/CallChainForm.aspx/a18/b3	ROS8423...		1.7 ms	0.0 µs
		/CallChain/CallChainForm.aspx/a0/b3	ROS8423...		1.6 ms	0.0 µs
		/CallChain/CallChainForm.aspx/a18/b7	ROS8423...		1.6 ms	0.0 µs
		/CallChain/CallChainForm.aspx/a0/b5	ROS8423...		1.6 ms	0.0 µs
		/CallChain/CallChainForm.aspx/a15/b4	ROS8423...		1.6 ms	0.0 µs
		/CallChain/CallChainForm.aspx/a14/b6	ROS8423...		1.6 ms	0.0 µs
		/CallChainForm.aspx	ROS8423...		1.6 ms	15.6 ms
		/CallChain/CallChainForm.aspx/a17/b5	ROS8423...		1.6 ms	15.6 ms
		/CallChain/CallChainForm.aspx/a18/b2	ROS8423...		1.6 ms	0.0 µs
		/CallChain/CallChainForm.aspx/a14/b1	ROS8423...		1.6 ms	0.0 µs
		/CallChain/CallChainForm.aspx/a15/b1	ROS8423...		1.6 ms	0.0 µs
		/CallChain/CallChainForm.aspx/a18/b0	ROS8423...		1.6 ms	0.0 µs
		/CallChain/CallChainForm.aspx/a15/b6	ROS8423...		1.5 ms	0.0 µs
		/CallChain/CallChainForm.aspx/a16/b2	ROS8423...		1.5 ms	15.6 ms
		/CallChain/CallChainForm.aspx/a18/b5	ROS8423...		1.5 ms	0.0 µs
		/CallChain/CallChainForm.aspx/a14/b3	ROS8423		1.5 ms	0.0 µs

After:

Status	Chart	Server Request	Probe	Latency % Over Thresh...	Latency	CPU (Avg)
		/CallChain/CallChainForm.aspx/*/b0	ROS84238...		1.2 ms	831.3 µs
		/CallChain/CallChainForm.aspx/*/b5	ROS84238...		1.1 ms	1.2 ms
		/CallChain/CallChainForm.aspx/*/b3	ROS84238...		1.1 ms	979.6 µs
		/CallChain/CallChainForm.aspx/*/b6	ROS84238...		1.1 ms	930.6 µs
		/CallChain/CallChainForm.aspx/*/b7	ROS84238...		1.1 ms	1.4 ms
		/CallChain/CallChainForm.aspx/*/b2	ROS84238...		1.1 ms	1.3 ms
		/CallChain/CallChainForm.aspx/*/b8	ROS84238...		1.1 ms	1.4 ms
		/CallChain/CallChainForm.aspx/*/b1	ROS84238...		1.1 ms	1.5 ms
		/CallChain/CallChainForm.aspx/*/b4	ROS84238...		1.1 ms	1.4 ms

For server request URIs that have been modified by the automatic URI collapsing feature, each associated call profile retains the original, uncollapsed, URI. You can view this value in the Original URI field in the Details pane of the Call Profile view.

<urireplacepattern> element

Purpose

Used to reduce the number of unique server requests by replacing many server requests with one simplified server request URI that aggregates them. Uses regular expression pattern matching. See "Configuring URI Truncation and Mapping" on page 183.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
enabled	true false	false	Enables uri pattern replacement.	No
pattern value	s/string/string/	If enabled there are two default patterns defined for you.	The syntax for the pattern value is s/search_pattern/replace_pattern/. If / is used in the pattern then the character # should be used instead of / as the separator. Patterns are applied to all server requests and are applied in the order they are specified in probe_ config.xml.	No

Elements

Number of Occurrences	1
Parent Elements	probeconfig, symbols
Child Elements	none

Example

```
<symbols maxuri="" maxsql="">
  <urireplacepattern enabled="true">
  <pattern value="s/TestService1/CommonService/"/>

  <pattern value="s/TestService2/CommonService/"/>
  </urireplacepattern>
</symbols>
```

<url> element

Purpose

Enables configuring which web pages are included or excluded from client monitoring.

Attributes

Attributes	Valid Values	Default	Description
name	/CallChain.*	include every page	Specifies which web pages to include or exclude from client monitoring, **Note:** Regular expressions can be used.

Changes to these attribute settings are applied dynamically; you do not need to restart the application or the probe.

Elements

Number of Occurrences	1
Parent Elements	filter
Child Elements	none

Example

```
<filter type="include">
    <url name=".*\.aspx" />
</filter>
```

<vmware> element

Purpose

Controls the ability to adjust timestamps to be more accurate when running in a VMware environment.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
attempttime stampadjustments	true false	false	Enables time stamp adjustments in VMware environments.	No
useworkaround	true false	false	If you encounter negative latency issues when running the .NET Agent on a VMware guest with the attempttimestampadjustments attribute set to true you should set this attribute to true. When this attribute is set to true the .NET Agent will use an alternative call to get the VMware host timestamps to workaround the negative latency issue.	No
disableperfcounters	true false	false	Set this option to true if the .NET Agent causes IIS worker process to crash in a VMWare environment.This is a workaround for a Microsoft-VMWare environment problem related to accessing perfmon counters in certain VMWare environments.	Yes

Elements

Number of Occurrences	1
Parent Elements	probeconfig
Child Elements	none

Example

```
<vmware attempttimestampadjustments="false"/>
```

<webserver> element

Purpose

Specifies the local Web server properties for communication with the probe.

Attributes

Attributes	Valid Values	Default	Description	Requires Application Restart
start	number	35000	Starting port for webserver.	Yes
end	number	35100	Ending port for webserver.	Yes
ipaddress	IP address		Local ip address to run webserver on.	Yes

Example

```
<webserver start="35000" end="35100" ipaddress="16.255.18.99"/>
```

<ws> element

Purpose

Controls Web services correlation sampling.

Attributes

None.

Elements

Number of Occurrences	1
Parent Elements	<xvm>
Child Elements	<sample>

Example

```
<xvm><ws><sample method="percent" rate="50"/></ ws ></xvm>
```

<xvm> element

Purpose

Controls the cross VM settings.

Attributes

None.

Elements

Number of Occurrences	1
Parent Elements	probeconfig, process, or appdomain
Child Elements	<ws>

Example

```
<xvm></xvm>
```

Chapter 7: Advanced .NET Agent Configuration

Instructions are provided for advanced configuration of the .NET Agent. Advanced configuration is intended for experienced users with in-depth knowledge of this product. Use caution when modifying any of the Diagnostics components' properties.

This chapter includes:

- "Time Synchronization for .NET Agents Running on VMware" on the next page
- "Customizing the Instrumentation for ASP.NET Applications" on the next page
- "Discovering the Classes and Methods in an Application" on page 175
- "Controlling Which HP Software Products the Agent can Work With" on page 177
- "Configuring Support for MSMQ BasedCommunication" on page 179
- "Configuring Latency Trimming and Throttling" on page 179
- "Configuring Depth Trimming" on page 182
- "Configuring URI Truncation and Mapping" on page 183
- "Capturing HTTP Server Requests Based on Query Parameters" on page 184
- "Configuring the .NET Agent for Lightweight Memory Diagnostics" on page 186
- "Limiting Exception Stack Trace Data" on page 188
- "Configuring Thread Stack Trace Sampling" on page 190
- "Disabling Logging" on page 192
- "Overriding the Default Probe Host Machine Name" on page 192
- "Listing the Probes Running on a Host" on page 193
- "Authentication and Authorization for .NET Profilers" on page 194
- "Configuring Consumer IDs" on page 195
- "Configuring SOAP Fault Data" on page 199
- "Collecting Additional Probe Metrics or Modifying Probe Metrics" on page 199

Time Synchronization for .NET Agents Running on VMware

.NET Agents running in VMware hosts have additional time synchronization requirements. For agents running in a VMware guest, time must be synchronized between the VMware guest and the underlying VMware host. If time is not synchronized properly, the Diagnostics UI could display inaccurate metrics or no metrics at all from a probe running in a VMware guest.

Time should be synchronized according to the recommendations given in the VMware whitepaper on timekeeping (`http://www.vmware.com/pdf/vmware_timekeeping.pdf`) in the section "Synchronizing Hosts and Virtual Machines with Real Time." In summary, VMware Tools must be installed in each VMware guest operating system that hosts a Diagnostics probe and the time synchronization option in VMWare Tools should be turned on. Note that this option in VMware Tools will only work if the guest operating system time is initially set earlier than that of the VMware host. For instructions on how to install VMware Tools, see the "Basic System Administration" document for VMware ESX Server. In addition, if any non-VMware time synchronization software (such as Network Time Protocol) is used, it should be run in the VMware ESX server service console.

Customizing the Instrumentation for ASP.NET Applications

When the .NET Agent is installed, the **ASP.NET.points** file is created with the standard instrumentation that the agent applies to all ASP.NET processing on the monitored server.

You must create application-specific instrumentation points to capture performance metrics for the business logic that has been implemented through application-specific classes and methods. The application-specific instrumentation points must be stored in a custom capture points file that can be associated with the application using the attributes in the **<probe_install_dir>/etc/probe_config.xml** file. If the application was auto-detected during the installation or during a rescan of IIS, a custom capture points file was automatically created for the application at the same time.

> **Note:** If you do not know the classes and methods in an application that you want to monitor, you can use the Reflector tool that was installed with the .NET Agent to analyze the .dll files in the application and discover the classes and methods. See "Discovering the Classes and Methods in an Application" on page 175 for instructions on using Reflector.

To let the .NET Agent know that you want the instrumentation points in a custom capture points file to apply to an application, you must update the **points** attribute of the **appdomain** element in the **probe_config.xml** file.

To associate a custom capture points file with an application:

1. Create a capture points file with the instrumentation for the application specific classes. To create a capture points file, copy an existing capture points file in the **<probe_install_dir>/etc** directory.

   > **Note:** If the application was auto-detected during the installation or during a rescan of IIS, a

capture points file already exists for the application with some or all of the points file entries commented out.

2. Customize the capture points file by adding instrumentation points so that the agent captures custom business logic for the applications.

   The following example illustrates how to modify the capture points file so that the agent captures IBuySpy custom code:

```
[IBuySpy Callee]
class = !IBuySpy.*
method = !.*
signature =
scope =
ignoreScope =
layer = Custom.IBuySpy
```

   For more information about instrumentation, see "Custom Instrumentation for .NET Applications" on page 59

3. Update the configuration of the .NET Agent probe in **probe_config.xml** to ensure that the modified capture points file is properly referenced.

   Within the ASP.NET **<process>** tag add an **<appdomain>** tag for the application. Include the **<points>** tag with the **file** attribute and the **enabled** attribute. See "Virtual Directories (AppDomains) Under Different IIS Paths with the Same Names" on the next page for more examples.

```
<appdomain name="1/ROOT/your_app_name" website="Default Web Site"
enabled="true">
  <points file="DefaultWebsite-your_app.capture points"/>
</appdomain>
```

   The example below illustrates this step. A custom capture points file has been created for the MSPetsShop application. The file has been named **MSPetShop.points**. The **<appdomain>** tag for the application, and the capture points file were added to the ASP.NET **<process>** tag in the **probe_config.xml** file. Note that the IIS path is included in the appdomain tag.

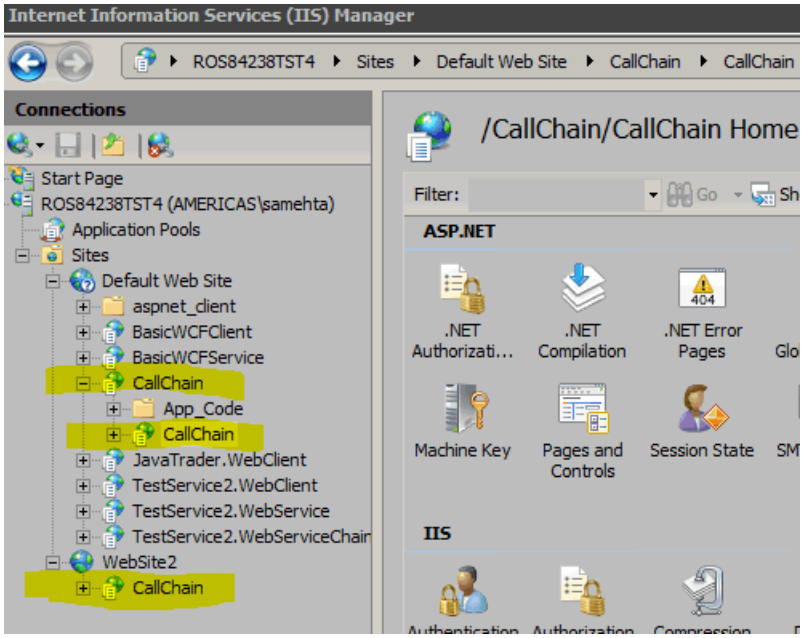
```
<?xml version="1.0" encoding="utf-8"?>

<probeconfig>
  <id probeid="" probegroup="Umatilla"/>

  <credentials username="" password=""/>
  <profiler authenticate=""><authentication username=""
password=""/></profiler>
```

```
  <diagnosticsserver url="http://issaquah:2006"/>
  <mediator host="issaquah" port="2612"/>
  <webserver start="35000" end="35100"/>
  <modes am="true"/>

  <instrumentation><logging level="" threadids="no"/></instrumentation>

  <lwmd enabled="true" sample="1m" autobaseline="1h" growth="10" size="10"/>

  <process name="ASP.NET", enablealldomains="false">
    <logging level=""/>
    <points file="ASP.NET.points"/>
    <appdomain name="1/ROOT/MSPetShop" website="Default Web Site"
enabled="true">
      <points file="DefaultWebsite-MSPetShop.points"/>
    </appdomain>
  </process>
</probeconfig>
```

4. Restart IIS as instructed in "Discovery and Standard Instrumentation" on page 41.

Virtual Directories (AppDomains) Under Different IIS Paths with the Same Names

You can distinguish two or more appdomains on the same IIS server which have the same name. Consider the configuration below where there are 3 virtual directories (AppDomains) with the name CallChain.



In the **probe_config.xml** file you can distinguish the AppDomains by including the IIS configuration path.

The configuration for the 3 CallChain applications in the example above would be as follows:

```
<appdomain enabled="false" name="1/ROOT/CallChain/CallChain" website="Default
Web Site">
  <points file="Default Web Site-CallChain-CallChain.points" />
</appdomain>
<appdomain enabled="false" name="1/ROOT/CallChain" website="Default Web Site">
  <points file="Default Web Site-CallChain.points" />
</appdomain>
<appdomain enabled="false" name="2/ROOT/CallChain" website="WebSite2">
  <points file="WebSite2-CallChain.points" />
</appdomain>
```

The resultant probes are distinguished using the IIS path and are displayed in the Enterprise UI as: 1ROOTCallChain.NET, 1ROOTCallChainCallChain.NET, 2ROOTCallChain.NET

Backward Compatibility with Pre-9.01 Releases

For the sake of backward compatibility, the 9.01 or later version of the agent will be able to read and process versions of the probe configuration earlier than 9.01 for ASP.NET AppDomains. The 'earlier' format is shown in the example below:

```
<appdomain name="CallChain">
  <points file="CallChain.points" />
</appdomain>
```

If you use the earlier format, then the behavior of the agent will revert to the previous version's behavior.

- All AppDomains with name "CallChain" (in this example) will be enabled or disabled simultaneously.
- All CallChain probe instances will be consolidated on the server into one probe.
- Trend lines for probes and server requests should continue from previous versions.

It is recommended that you do NOT use the earlier format of configuration where backward compatibility (such as trend lines) is not required.

For an appdomain configured using the earlier format, if the new behavior is desired, the "old" format entry should be deleted from the probe_config.xml file. Then run **Rescan ASP.NET Applications** from the start menu on the probe system. This will result in the addition of AppDomain entries with the new format, allowing you to distinguish different probes on the same IIS server with the same name.

The upgrade install will retain the earlier version of the appdomain configuration and modify probe_ config.xml to add the new format configuration for any unlisted AppDomains.

Discovering the Classes and Methods in an Application

To monitor the performance of an application that you are not familiar with, use the Reflector automatic discovery tool that is installed with the .NET Agent to find the classes and methods in the application that you want to add to the instrumentation used by a probe. The Reflector executable is located at **<probe_install_dir>\bin\reflector.exe**.

To discover classes and methods using Reflector:

1. Locate the installation directory for the application that you want to monitor.
2. Locate the folder in the application installation directory where the .dll files are stored.
3. Open a command prompt and change the directory to the folder where the .dll files for the application are stored.
4. Run the Reflector against all of the .dll files and .exe files in the current directory by executing the following the command at the command prompt:

   ```
   <probe_install_dir>\bin\Reflector.exe
   ```

   You can limit the Reflector to certain .dll and .exe files by adding additional parameters to the command. The following example shows another way to enter the command in the previous example:

   ```
   <probe_install_dir>\bin\Reflector.exe *.dll *.exe
   ```

   This command explicitly tells the Reflector to check all of the .dll and .exe files in the target directory.

   To limit the Reflector to specific files, you could enter the following:

   <probe_install_dir>\bin\Reflector.exe WorkHorse.dll Utility.dll

   This command explicitly tells the Reflector to check only the two .dll files specified.

   The following example shows the commands you might execute if you have an application called PetShop that has .dll files located in a bin folder:

   ```
   C:\>cd "c:\Program Files\Microsoft\PetShop\Web\bin"

   C:\Program Files\Microsoft\PetShop\Web\bin>
   C:\MercuryDiagnostics\".NET Probe"\bin\Reflector.exe
   ```

5. The Reflector displays a report of the assemblies, namespaces, classes, and methods found in the .dll files that you specified.

```
--------------------------------------------------------------------------------
Assemblies:
--------------------------------------------------------------------------------
C:\Program Files\Microsoft\PetShop\Web\bin\PetShop.BLL.dll
C:\Program Files\Microsoft\PetShop\Web\bin\PetShop.DAL.dll
C:\Program Files\Microsoft\PetShop\Web\bin\PetShop.Web.dll


--------------------------------------------------------------------------------
Namespaces:
--------------------------------------------------------------------------------
(8  classes)  PetShop.BLL
(6  classes)  PetShop.DALFactory
(17 classes)  PetShop.Web
(11 classes)  PetShop.Web.Controls
(2  classes)  PetShop.Web.ProcessFlow
(1  classes)  PetShop.Web.WebComponents


--------------------------------------------------------------------------------
(8 classes)  Namespace: PetShop.BLL
--------------------------------------------------------------------------------
PetShop.BLL.Account (10 Methods)
    Equals              System.Boolean(System.Object)
    Finalize            System.Void()
    GetAddress          PetShop.Model.AddressInfo(System.String)
    GetHashCode         System.Int32()
    GetType             System.Type()
    Insert              System.Void(PetShop.Model.AccountInfo)
    MemberwiseClone     System.Object()
    SignIn              PetShop.Model.AccountInfo(System.String,System.String)
    ToString            System.String()
    Update              System.Void(PetShop.Model.AccountInfo)

PetShop.BLL.Cart (17 Methods)
    Add                   System.Void(System.String)
    Equals                System.Boolean(System.Object)
    Finalize              System.Void()
    get_Count             System.Int32()
    get_Item              PetShop.Model.CartItemInfo(System.Int32)
    get_Total             System.Decimal()
    GetCartItems          System.Collections.ArrayList()
    GetEnumerator         System.Collections.IEnumerator()
    GetHashCode           System.Int32()
    GetInStock            System.Int32(System.String)
    GetOrderLineItems     System.Collections.ArrayList()
    GetType               System.Type()
    MemberwiseClone       System.Object()
```

Note: You can redirect the output from the Reflector to a file, as shown in the following example:

<probe_install_dir>\bin\Reflector.exe sys*.dll > <report_name>.txt

The output from Reflector is redirected to the file that you specify.

Use the information in the report to customize the instrumentation for the application, as described in "Customizing the Instrumentation for ASP.NET Applications" on page 171.

Controlling Which HP Software Products the Agent can Work With

The .NET Agent can be set in different modes for the following:

- Monitoring applications from development through pre-production testing and into production.
- Use with other HP Software products.
- Use as a standalone Diagnostics Java Profiler not reporting to a server or to other HP Software products.

The mode the .NET Agent works in is determined by the **<modes>** element set in the **<probe_install_dir>/etc/probe_config.xml** file.

The **<modes>** element is also used in determining usage against the license capacity (see "License Information Based on Currently Connected Probes" in the HP Diagnostics Server Installation and Administration Guide). For Diagnostics there are two types of LTUs (License to use):

- AM - When using of the product in an enterprise mode, typically in a production environment.
- AD - When using the product in a pre-production load testing environment with probes in LoadRunner or Performance Center runs.

The value of the <modes> element is initailly set at the time you install the .NET agent. See "Installing .NET Agents " on page 18.

To change the value of the <modes> element you can edit the probe_config.xml file. Or you can re-run the .NET Agent installer and use the Change option to set the mode to Diagnostics Profiler Mode (PRO), Application Management/Enterprise Mode for Diagnostics (Enterprise) and/or TransactionVision (TV) or Diagnostics Mode for LoadRunner/Performance Center (AD).

> **Note:** To use the standalone Diagnostics Profiler for .NET in enterprise mode or integrated with other HP Software products, contact HP Software Customer Support to purchase HP Diagnostics.

To see Diagnostics data in the user interface of the interfacing HP Software products, you must perform additional configuration steps. See BSM-Diagnostics Integration Guide and the LoadRunner/Peformance Center-Diagnostics Integration Guide.

The sections that follow provide instructions for configuring each product mode of the **<modes>** element (see also "<modes> element" on page 133).

PRO Mode - Diagnostics Profiler for .NET

When PRO mode is set, the agent gathers performance metrics and presents them in the standalone Diagnostics Profiler for .NET user interface which is made available through a URL on the agent host.

In this mode the profiler is always collecting data even when the profiler UI is not in use. This mode can be combined with other modes.

PRO mode is not used in determining usage against license capacity.

Enterprise Mode

When configured in Enterprise mode, the agent works with HP Software products such as BSM, LoadRunner, Performance Center, and as the full Diagnostics enterprise product. It will capture data for LoadRunner/Performance Center runs in a separate database as well as capture data outside of LoadRunner/Performance Center runs.

Both AD and AM modes will override this mode.

In Enterprise mode data will also be sent to the Diagnostics .NET Profiler. If the PRO mode is set along with Enterprise mode then the .NET Agent will collect data continuously for the profiler even if the profiler UI is not in use. If PRO mode is not set then the agent will not start collecting data until the profiler UI is started.

Enterprise mode is the default for .NET Agents (if you don't specify AD or AM mode). In Enterprise mode the agents are counted against the AM license capacity.

AM Mode

In AM mode the .NET agent will capture all instrumentation data. You can set `AM` mode to protect an agent in a production BSM deployment from accidently being included in a LoadRunner or Performance Center run. In AM mode, the agent is not listed as an available agent in LoadRunner or Performance Center.

Agents in AM mode will always be counted against the AM license capacity.

AM mode supersedes all other modes except for AD.

AD Mode

In AD mode the .NET agent will only capture data during runs from LoadRunner/Performance Center and the results will be stored in a specific Diagnostics database for that run, for example, Default Client:21.

When the agent is in this mode it will not use resources or send any data to the server unless the probe is part of a LoadRunner/Performance Center run.

AD mode supersedes all other modes. So for example, if AD mode and any other modes are set then the mode will be set to AD.

See the BSM-Diagnostics Integration Guide and the LoadRunner/Peformance Center-Diagnostics Integration Guide for more information.

Use this mode to prevent an agent in a QA environment from using additional resources and continually report data to the Diagnostics console dataset when a load test is not running.

Another advantage of running a probe in AD mode is that probes in AD mode will only be counted against AD license capacity when the probe is running in a LoadRunner or Performance Center test run. For example if you have 20 agents installed in LoadRunner/Performanace Center AD mode but only 5 are in a run, then only 5 are counted against AD license capacity.

TV mode

This mode will send events to Transaction Vision. This mode can be combined with other modes. TV mode is not used in determing usage against HP Diagnostics license capacity.

Note about AD Mode and Enterprise Mode

The .NET agent gets notified of LoadRunner/Performance Center runs by the Diagnostic Mediator.

If LoadRunner/Performance Center starts testing an instrumented application that is not running, for example, a web application getting hit the first time, then when the application starts executing the Diagnostics agent will not be notified of the run. This is because the agent will not have had enough time to get initialized and start listening to the mediator for this notification.

To work around this problem, the .NET agent needs to be "primed"(initialized) by a call to the web application before a LoadRunner/Performance Center run is started. This initializes the web application's process (worker process) and the probe so that it is ready to accept run information from the mediator.

Configuring Support for MSMQ BasedCommunication

To configure the .NET Agent to support MSMQ based communication, include the **msmq.points** file in the scope of the appdomain as shown in the example excerpt from a **<probe_install_dir>/etc/probe_config** file:

```
<process name="SimplestQueuingSender">
<points file="msmq.points"/>
<modes enterprise="true"/>
</process>
```

Configuring Latency Trimming and Throttling

When the .NET Agent determines that it is running out of resources because the Diagnostics Server is not keeping up with the amount of data that the probes are capturing, the agent can automatically reduce the number of methods the probe captures using a process called *latency trimming*. By default, latency trimming is enabled so that the probe's work load can be adjusted as necessary.

When latency trimming is enabled, the .NET Agent trims the number of methods captured by a probe by ignoring methods with a total latency below a certain minimum latency threshold. The idea behind trimming is that it is better to miss capturing methods with lower latency that are less likely to be of interest than to allow the probe to bog down or stop running. Trimming allows the probe to continue to run so that it can capture the more interesting methods with higher latencies.

Note: Because of threading and buffering, partial information about a method that was trimmed can be transmitted to the Diagnostics Server. When the Diagnostics Server detects that it received only partial information for a method, it issues a warning message. You should ignore these warning messages unless you expected that the information for all methods was to be captured.

Note:

- Latency trimming and throttling are ignored by the Profiler user interface.
- The Diagnostics Server can be configured to apply additional trimming of the probe's data which will affect the granularity of the data shown by the Diagnostics user interface.

Disabling Latency Trimming

By default, trimming is enabled for the .NET Agent. To disable trimming you must change the configuration.

To disable Latency Trimming:

Add the **latency** tag to the **<probe_install_dir>/etc/probe_config.xml** configuration file, as shown in the following example:

```
<trim>
  <latency enabled="false" />
</trim>
```

The attribute of the latency element that turns on latency trimming is **enabled**. Latency trimming is enabled when **enabled** is set to `true`. When **enabled** attribute is set to `false`, latency trimming is disabled. The default value for this attribute is `true`.

For a description of attributes and elements of the **latency** element, see "Understanding the .NET Agent Configuration File " on page 85

Enabling Latency Trimming

By default, trimming is enabled for the .NET Agent. If you subsequently disabled trimming, you must change the configuration to enable it once more.

To enable Latency Trimming:

Change the value of the enabled attribute of the **latency** element in the **<probe_install_dir>/etc/probe_config.xml** configuration file, as shown in the following example:

```
<trim>
  <latency enabled="true" />
</trim>
```

The attribute of the latency element that turns on latency trimming is **enabled**. Latency trimming is enabled when **enabled** is set to **true**. When **enabled** attribute is set to **false**, latency trimming is disabled. The default value for this attribute is **true**.

For a description of attributes and elements of the **latency** element, see "Understanding the .NET Agent Configuration File " on page 85

Setting Latency Trimming Thresholds

By default, the latency trimming thresholds are set so that those methods with a latency less than 2 ms are trimmed, and those methods with a latency greater than 100 ms are never trimmed.

You can set the minimum trimming threshold by adjusting the value of the **min** attribute. You can set the maximum trimming threshold by adjusting the value of the **max** attribute. These attributes are specified in the **latency** element in the **<probe_install_dir>/etc/probe_config.xml** configuration file.

```
<trim>
  <latency enabled="true" min="50" max="100" />
</trim>
```

The attributes of the latency element that control the trimming thresholds are:

- **min**

  Sets the minimum latency threshold. When latency trimming is enabled, methods with a latency less than or equal to the value of this attribute are trimmed. If you do not specify a value for this attribute, the default value of `2 ms` is used.

  The lower the value of the **min** attribute the greater the chance that the performance of the application will be adversely impacted. A lower value means that fewer methods are trimmed because more low-latency methods are captured.

  If the information for all methods must be captured, disable latency trimming by setting **latency enabled** equal to false.

- **max**

  Sets the maximum latency threshold. When latency trimming is enabled, methods with a latency greater than or equal to the value of this attribute are never to be trimmed. The default value for this attribute, if you do not specify a value, is `100ms`.

For a description of the attributes and elements of the **latency** element, see "Understanding the .NET Agent Configuration File " on page 85

Configuring Latency Trimming Throttling

Latency trimming is throttled by default. When throttling is enabled, the amount of trimming that is done is automatically adjusted based on the percentage of the probe resources that are being used up by the Diagnostics Server processing backlog.

Without throttling, the methods that fall below the minimum method latency threshold are always trimmed.

If the percentage resources used by the probe increases above a set throttling increment threshold, the effective trimming threshold is incremented so that methods with higher latency are trimmed. If the percentage of probe resources used increases above the threshold again, the effective trimming threshold is incremented once more so that methods with even higher latency are trimmed. If the percentage of probe resources used drops below the throttling decrement threshold, the effective trimming threshold is decremented so that the methods with lower latencies are captured once more.

The effective trimming threshold cannot be incremented above the maximum method latency threshold, and it cannot be decremented below the minimum method latency threshold.

Below is an example of the **latency** element in the **probe_config.xml** configuration file that includes the throttling attributes:

```
<trim>
  <latency enabled="true" min="50" max="100"
  throttle="true" incrementthreshold="75"
  decrementthreshold="50" increment="2"/>
</trim>
```

The attributes of the **latency** element that control throttling are:

- **throttle**

  Throttling is enabled when this attribute is set to `true`. When this attribute is set to `false` throttling is disabled. The default value for this attribute is `true`.

- **increment**

  Sets the amount that the effective trimming threshold is incremented when the percentage of probe resources used exceeds the **incrementthreshold**. Sets the amount that the effective trimming threshold is decremented when the **decrementthreshold** is crossed. The default value for this attribute is `2` ms.

- **incrementthreshold**

  When the percentage of probe resource usage rises to the value of this attribute or higher, throttling is triggered so that the effective trimming threshold is incremented. The default value for this attribute is **75** percent.

- **decrementthreshold**

  When the percentage of probe resource usage falls to the value of this attribute or lower, throttling is triggered so that the effective trimming threshold is decremented. The default value for this attribute is **50** percent.

For a description of the attributes and elements of the **latency** element, see "Understanding the .NET Agent Configuration File " on page 85.

Configuring Depth Trimming

The .NET Agent can automatically reduce the number of methods that it captures using a process called *depth trimming*. When the Diagnostics Server is not keeping up with the amount of data that the probe is capturing, the probe can use depth trimming to help prevent it from running out of resources. By default, depth trimming is enabled.

Note: Depth trimming is ignored by the Profiler user interface.

When depth trimming is enabled, the .NET Agent trims the number of methods captured by ignoring methods that are called at a stack depth that is greater than the maximum stack depth threshold. Those that are called at a stack depth less than or equal to the stack depth threshold are captured. The idea behind trimming is that it is better to miss capturing methods further down in the call stack, that are less

likely to be of interest, so that the probe is able to continue to run and is able to capture the more interesting methods that occur higher in the call stack.

For example, if the stack depth threshold is 3, and the following method calls are made:

```
/login.do calls a() calls b() calls c()
```

where only the `/login.do`, `a`, and `b` methods are captured, and method c is trimmed.

Below is an example of the **depth** element in the **probe_config.xml** configuration file that includes the trimming attributes:

```
<trim>
  <depth enabled="true" depth="10" />
</trim>
```

The attributes of the **depth** element that control trimming are:

- **enabled**

  Depth trimming is enabled when this attribute is set to `true`. When this attribute is set to `false` depth trimming is disabled. The default value for this attribute is `true`.

- **depth**

  Sets the threshold that are used for depth trimming. Methods that are called at or below the value of this attribute are trimmed when depth trimming has been enabled. The default value for this attribute is **25**.

  Setting **depth** to a lower value can significantly reduce the overhead of capture. For a description of the attributes and elements of the **depth** element, see "Understanding the .NET Agent Configuration File " on page 85.

Configuring URI Truncation and Mapping

Any HTTP/S server request URI can be transformed before being reported by the probe. This transformation is based on regular expression matching and replacement controlled by the **urireplacepattern** element in the **probe_config.xml** configuration file. It is turned off by default.

This can be useful when you are seeing too many server requests and you want to replace many server request URIs with one simplified server request URI that aggregates them.

Caution: Overuse of this feature will impact performance.

An example is shown below:

```
<symbols maxuri="" maxsql="">
  <urireplacepattern enabled="true">
    <pattern value="s/TestService1/CommonService/"/>
```

```
    <pattern value="s/TestService2/CommonService/"/>
  </urireplacepattern>
</symbols>
```

The syntax used for the pattern value is s/search_pattern/replace_pattern/.

The search_pattern and replace_pattern should be enclosed in /. If / is used in the pattern then the character # should be used instead of / as the separator.

The patterns are applied to all server requests and are applied to the uri in the order they are specified in the probe_config.xml file.

If urireplacepattern is enabled, then two default patterns are configured by default.

The first of these default patterns is used to trim server requests that contain a ; or /!. All content after these tokens is removed from the server request.

The pattern used is : s#(;|/?\\!).*$##"

The second of these default patterns replaces loading of images, pdfs and docs with a fixed token ("/Static Content").

The pattern used is:

s#(?<word1>^.*)(/.*\\.js|css|jpg|gif|png|pdf|html|doc|docx)#${word1}/Static Content#

Both of these patterns can be customized.

Capturing HTTP Server Requests Based on Query Parameters

An HTTP/S server request can be named based on its query parameters. This allows the probe to report more granular metrics for a particular server request.

By default, query parameters are ignored when monitoring a particular server request. To specify that a server request be created based on a particular query parameter, use the **httpcaptureparams** element in the **probe_config.xml** configuration file. Multiple parameters can be specified.

An example is shown below:

```
<httpcaptureparams enabled="true">
  <param name="Genre"/>
  <param name="accounttype"/>
</httpcaptureparams>
```

A server request is created for each server request that includes the Genre parameter and accounttype parameters:

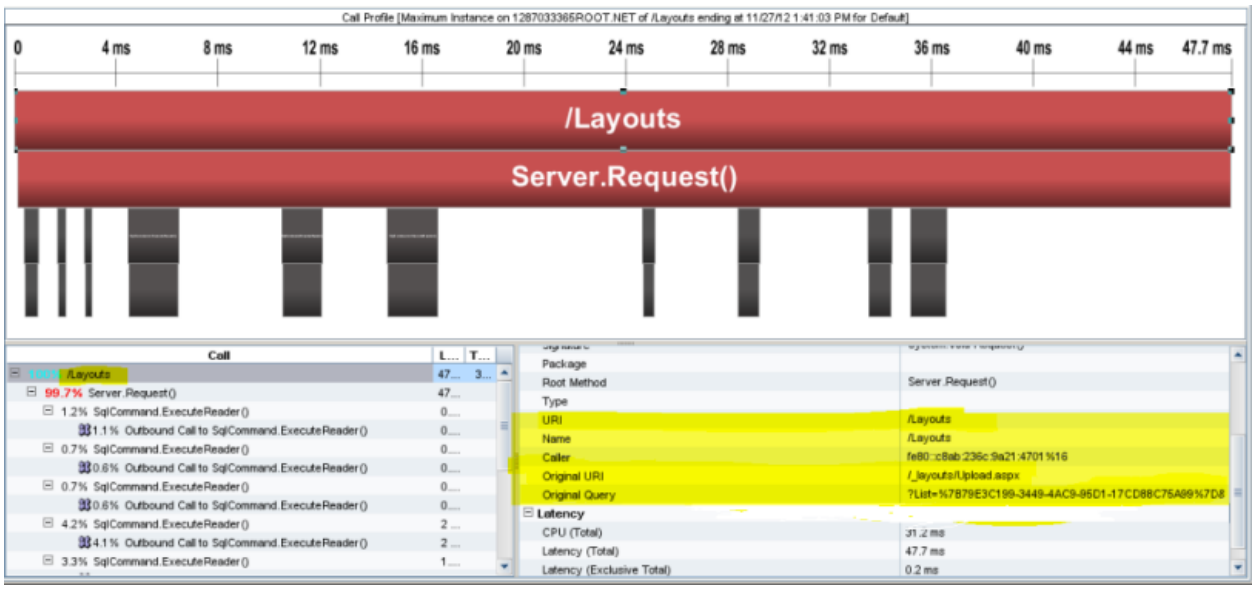
Table

Status	Chart	Server Request	Probe	Laten... Over ...	Latency	CPU (A...	Throu...	Excep...	Info
		/MVC3/MusicStore/	2ROO...		107.2 ms	109.2 ms	12 / hr	0	
		/MVC3/MusicStore/Account/LogOn?ReturnUrl=/MVC3/MusicStore/S...	2ROO...		310.9 ms	312.0 ms	12 / hr	0	
		/MVC3/MusicStore/ShoppingCart/AddToCart/241	2ROO...		1.3 s	1.3 s	12 / hr	0	
		/MVC3/MusicStore/Store/Browse	2ROO...		64.5 ms	50.7 ms	192 / hr	0	
		/MVC3/MusicStore/Store/Browse?Genre=Metal	2ROO...		62.4 ms	62.4 ms	12 / hr	0	
		/MVC3/MusicStore/Store/Browse?Genre=Pop	2ROO...		52.5 ms	46.8 ms	24 / hr	0	
		/MVC3/MusicStore/Store/Browse?Genre=Reggae	2ROO...		56.1 ms	41.6 ms	36 / hr	0	
		/MVC3/MusicStore/Store/Browse?Genre=Rock	2ROO...		72.2 ms	46.8 ms	24 / hr	0	
		/MVC3/MusicStore/Store/Details/241	2ROO...		426.5 ms	405.6 ms	12 / hr	0	

The **httppcaptureparams** element can also be used to capture the original URI of the server request. To capture the unmodified server request, set the **capturequerystring** argument to true:

```
<httpcaptureparams enabled="false" capturequerystring="true">
  <param name="Genre"/>
 </httpcaptureparams>
```

The captured query string is displayed in the call profile (SR instance) as shown below:



Note: Avoid using a session parameter or highly unique URI value because of the impact to overhead and data storage.

Configuring the .NET Agent for Lightweight Memory Diagnostics

The Lightweight Memory Diagnostics (LWMD) feature refers to the ability to capture and analyze usage data that relates to Collections. Specifically Collections refer to any class that implements either the **System.Collections.ICollection** or **System.Collections.Generic.ICollection** interfaces. Examples of such Collections are ArrayList, HashTable, DataView etc. The most common from of .NET memory leaks occur in Collections that are not properly maintained.

When the .NET Agent is installed, the default configuration for the .NET Agent probe is to have LWMD turned off. To enable the LWMD feature you must perform two modifications to the **probe_config.xml** file:

- You must enable the <lwmd> element (see "<lwmd> element" on page 127).
- You must add one or more references to the **Lwmd.points** file as described in the instructions below.

> **Note:** Enabling the probe to capture collections metrics could incur additional overhead on the host for an application.

To enable the capture of collection metrics for a process or for an AppDomain:

Add a **points** tag for the **Lwmd.points** file to either the **process** tag or to one or more **<appdomain>** tags in the **probe_config.xml** configuration file.

When you install the .NET Agent, the **Lwmd.points** file is installed in the **<probe_install_dir>/etc/** directory along with the **ASP.NET.points** and **ADO.points** files. The **Lwmd.points** file contains the instrumentation instructions needed to enable the capture of collection metrics.

To enable LWMD instrumentation for all enabled AppDomains that run under a process, you add the points tag to the process tag in the **probe_config.xml** configuration file. For example, to enable LWMD instrumentation for all enabled ASP.NET AppDomains:

```
<process name="ASP.NET", <enablealldomains="false">
  <points file="ASP.NET.points" />
  <points file="ADO.points" />
  <points file="Lwmd.points"/>
  <appdomain name="1/ROOT/your_app_name" website="Default Web Site
enabled="true">
    <points file="DefaultWebsite-your_app.capture points" />
  </appdomain>
</process>
```

To enable LWMD instrumentation for a specific enabled AppDomain that runs under a process, you add the points tag to an appdomain tag in the **probe_config.xml** configuration file. You can add the points

tag to one or more of the <appdomain> tags. For example, to enable LWMD instrumentation for the "your_app_name" AppDomain running in the ASP.NET process:

```
<process name="ASP.NET", <enablealldomains="false">
  <points file="ASP.NET.points" />
  <points file="ADO.points" />
  <appdomain name="1/ROOT/your_app_name" website="Default Web Site"
enabled="true">
    <points file="DefaultWebsite-your_app.capture points" />
    <points file="Lwmd.points"/>
  </appdomain>
</process>
```

To disable LWMD:

To disable the LWMD feature you must perform two modifications to the **probe_config.xml** file:

- Disable the <lwmd> element (see "<lwmd> element" on page 127).
- Delete the points tags for the **Lwmd.points** file from all process tags and from the appropriate <appdomain> tags.

  Without the LWMD points tags in the configuration file, the probe cannot locate the LWMD instrumentation instructions contained in the Lwmd.points file and so the probe will not instrument for Collection usage.

To control LWMD Instrumentation:

When the .NET Agent is installed, the default configuration for the Lwmd.points file contain instructions to instrument Collection usage in a wide range of assemblies, AppDomains, namespaces and classes. You can modify the your application's points file to narrow the scope of the Collections that you want to inspect. LWMD Instrumentation is implemented as Caller side Instrumentation, refer to "Caller Side Instrumentation" on page 68 for a description of how this instrumentation works.

Note: Narrowing the scope of LWMD instrumentation is a recommended best practice.

To narrow the scope of the Collections that you want to inspect perform the following steps:

1. Delete the points tags for the Lwmd.points file from the process tags and from the appropriate <appdomain> tags. This will remove the LWMD settings that specify a wide instrumentation scope.
2. Add an LWMD section to the points file for your process or AppDomain. As an example, to do this copy and paste the following into **your_app.points** file:

   ```
   [LWMD]
   keyWord = lwmd
   scope =
   ignoreScope =
   ```

3. Set the scope and ignoreScope Arguments in the LWMD section to narrow the scope of the Collections that you want to inspect. Example:

```
[LWMD]
keyWord = lwmd
scope = !my_namespace\..*
ignoreScope = !my_namespace.my_class1\..*
```

The example above instruments all the Collections that are constructed from the my_namespace namespace except for any Collections that are constructed from any method in the my_namespace.my_class1 class.

For LWMD Instrumentation there is an internal default value for ignoreScope that is unpublished and is always included with any value you enter. The default value includes namespaces and classes relating to the .NET Infrastructure that if instrumented would adversely affect the application, for example, !System.*, !Microsoft.*, and so on.

Limiting Exception Stack Trace Data

The agent collects exception data for exception throwing server requests and presents the information in the Diagnostics UI. The collected exception data can optionally include a stack trace.

Collecting stack trace data for all exceptions is usually undesirable however, because exception stack traces that are not of interest overload the display as well as the data collection and transfer operations. You can therefore limit the exception types for which stack trace data is collected. For example, filtering application server-based errors such as **System.Security.Authentication.AuthenticationException** would allow the stack traces to be used for more application-specific errors.

The stack trace data that is collected is controlled in three ways: limiting specific exception types, limiting the number of exceptions for which stack trace data is collected and limiting the size of the stack trace data.

Note: You can disable all stack trace collection by setting **captureexceptions enabled**="false" in the **probe_config.xml** file. By default, stack trace collection is enabled.

This section includes:

- "Limit Specific Exception Types" on the next page
- "Limit the Number of Exceptions per Server Request" on the next page
- "Limit the Size of the Stack Trace" on the next page
- "Example" on page 190

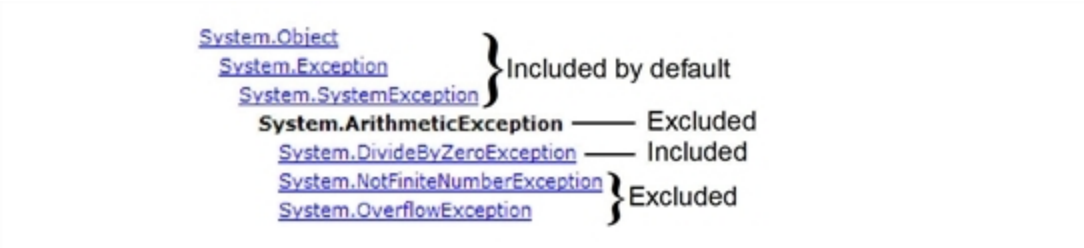
Limit Specific Exception Types

The exceptions for which stack trace data is collected is limited by setting the **exclude** and **include** properties in the **probe_config.xml** file as shown in the following example:

```
<exclude>
  <exceptiontype name="System.ArithmeticException"/>
</exclude>
<include>
  <exceptiontype name="System.DivideByZeroException"/>
</include>
```

Subtypes of any exception type specified to be excluded or included are also excluded or included, respectively, unless they are explicitly specified otherwise on the include or exclude list.

The following diagram shows which exception types are included and excluded based on the preceding example:



Changes to the **probe-config.xml** file take effect immediately; it is not necessary to restart the application.

Limit the Number of Exceptions per Server Request

By default, the .NET Agent probes collect stack trace data on only the first 4 exceptions encountered during a server request. If your application has more exceptions for which you want to view stack trace information, you can increase the value of the **max_per_request** property in the **probe_config.xml** file. As with all collected metrics, increased amounts of collected data place a higher load on the Diagnostics Server.

Limit the Size of the Stack Trace

By default, the captured stack trace data can be of any size. You can limit the size of the stack trace string to improve the readability of the Exceptions tab. Set the value of the **max_stack_size** property to the maximum stack trace string in the **probe_config.xml** file. As with all collected data, increased amounts of collected data place a higher load on the Diagnostics Server. By default, this property is set to 0 (zero) which means that the stack trace size is not limited.

Example

The following settings enable exception stack traces with a maximum stack trace string size of 2048.

```
<captureexceptions enabled="true" max_per_request="4" max_stack_size="2048">
  <exclude>
    <exceptiontype name="System.ArithmeticException"/>
  </exclude>
  <include>
    <exceptiontype name="System.DivideByZeroException"/>
  </include>
</captureexceptions>
```

Configuring Thread Stack Trace Sampling

When asynchronous thread sampling is enabled, you can see, in the Call Profile view, which methods were executed during long running fragments even if no instrumented methods were hit during this time. See the HP Diagnostics User Guide chapter on Call Profiles for a screen shot showing the additional nodes added based on thread sampling.

The <stacktracesampling> element in probe_config.xml enables and configures thread stack trace sampling. For more information about this element, see "<stacktracesampling> element" on page 152.

Example Thread Sampling Configurations

Use Case 1: You see a particular method that intermittently takes an exceptionally long time to complete. Since the method average execution time is relatively short, you do not want to add additional instrumentation to the methods callable from the method, because this would increase the overhead.

1. You enable stack trace sampling and configure the long method latency threshold to a value larger than the average execution time of the method, but shorter than the observed long running times.
2. The stack traces are collected only for methods running at least as long as the specified threshold value, thus incurring no overhead for most cases.
3. You examine the Call Profile for the long running instances of the Server Request and sees additional nodes revealed by stack trace sampling.

Example:

In production environment, a particular method has average latency about 170 milliseconds, but from time to time it takes 1.4 second for this method to complete. Most of the methods visible in Call Profiles for any fragment execute in about 550 milliseconds or less.

Since the method in question makes multiple calls to its callees, you do not want to instrument them. Instead,you enable sampling to find out what is the cause of long execution times. To minimize the overhead, you set tardymethodlatency value to 600 milliseconds. This ensures that most of the

methods will not get sampled at all, because they are likely to complete before this time elapses. However, any method running longer than this value, including our trouble-making method, will get sampled, once the method runs for 600 milliseconds (or longer) without making any calls to any of the instrumented methods.

You also set the value of stacktracesampling-rate to 100 milliseconds. Theoretically, this should give up to 8 samples for each method invocation that lasts 1.4 seconds ((1400 - 600) / 100).

Use Case 2: You see insufficient Call Profile info for all or some of the Server Requests, but are reluctant to add additional instrumentation because of the performance concerns or because of the need to restart the application.

1. You enable stack trace sampling, resets the long method latency to zero, and configures the sampling rate to balance the overhead and the amount of additional data.
2. The stack traces are collected for all methods
3. You examine the Call Profiles and see additional nodes revealed by stack trace sampling

Example: You prepare a custom application for deployment and sees that the default instrumentation provided with the Diagnostics probe does not work very well, because many Call Profiles contain very few methods which do not give any insight about the application specific behavior. You are reluctant to add additional instrumentation for all classes and methods belonging to her custom application, because of the performance and memory consumption concerns.

Assuming that a typical fragment that does not have sufficiently detailed call tree information runs in about 2 seconds, you select stacktracesampling-rate to be 200ms. This can give up to 10 stack traces per typical fragment. However, you do not want all the stack traces to be reported, because some of the methods visible in the stack traces can be very fast, and they do not substantially contribute to the fragment overall latency. After viewing the Call Profiles with the additional method nodes obtained from sampling, you make an informed decision about adding additional instrumentation points to the probe configuration in deployment.

Troubleshooting Thread Sampling Configurations

Why do I not see any new nodes in my Call Profile after I enabled stack trace sampling?

See if any of the following applies to your case:

Check if the last method visible in the Call Profile is an outbound call. Outbound calls do not get sampled by default.

- Try to reduce tardymethodlatency. It is possible that the last method visible in Call Profile makes calls that get trimmed, but they prohibit the sampling to kick in because there's never an inactive period of tardymethodlatency for the caller.
- Try to reduce stacktracesampling - rate. Perhaps your methods simply miss the opportunities to get sampled.
- Verify that the latency of the last visible method in Call Profile is not caused by running garbage collector. No .NET code runs during garbage collection, and this includes the stack trace sampling code.

What is the minimum value of stacktracesampling.rate I can use?

You can use any positive value, but please keep in mind that each platform will simply refuse to sample more frequently that it possibly can. The three factors playing a role here are: the minimum granularity of sleep() available, the timer resolution, and the time it actually takes to collect one set of samples. It is recommended to be higher than 20 ms.

What is the maximum value of stacktracesampling-tardymethodlatency I can use?

There is no limit. The usefulness of a high setting depends entirely on the latency of the server requests for the application. To get any results, you should plan for at least a few samples for each fragment you are concerned with, and even that may require tuning other sampling parameters as well.

Disabling Logging

You can disable application logging by changing the **logging level** tag of the ASP.NET process section of the **probe_config.xml** file, as shown in the following example:

```
<process name="ASP.NET">
  <logging level="off"/>
</process>
```

You can disable instrumentation logging by changing the **logging level** tag of the instrumentation section, as shown in the following example:

```
<instrumentation>
  <logging level="off" />
</instrumentation>
```

Overriding the Default Probe Host Machine Name

The **registered_hostname** property enables you to override the default host machine name that a probe uses to register itself with the Diagnostics Server in Commander mode. In situations where a firewall or NAT is in place or where your probe host machine has been configured as a multi-homed device, it might not be possible for the Diagnostics Server in Commander mode to communicate with the probe unless you override the default host machine name.

To override the default host machine name for a probe there is a three step process.

1. First, set the **registered_hostname** attribute, located in the .NET Agent **<diagnosticsserver>** element of the **probe_config.xml file**， to an alternate machine name or IP address that allows the Diagnostics Server in Commander mode to communicate with the Probe.

   For example:

   ```
   <diagnosticsserver url="http://localhost:2006/commander" registered_
   ```

```
hostname=" my_host_name "/>
```

2. Second, register the alternate machine name or IP address of the host with the .NET Metrics Agent. To do this, make a **metrics.agent.registered_hostname** entry in the **metrics.config** file. You can add the entry just under the metrics.systemgroup entry.

   For example:

```
metrics.systemgroup = Default
metrics.agent.registered_hostname = my_host_name
```

3. Finally, you must restart both the .NET Agent and the .NET Metrics Agent for this change to take effect.

Note:

- Setting the **registered_hostname** attribute because of a NAT or firewall is only an issue for a test environment where you are using LoadRunner, Performance Center, or Diagnostics Standalone.
- You need to set the **registered_hostname** attribute to deal properly with the use of the IIS Host Header technology.
- However, if you should set the registered_hostname in a production environment where you are using BSM or Diagnostics Standalone, the name that you specify is shown as the host name in System Health.

Listing the Probes Running on a Host

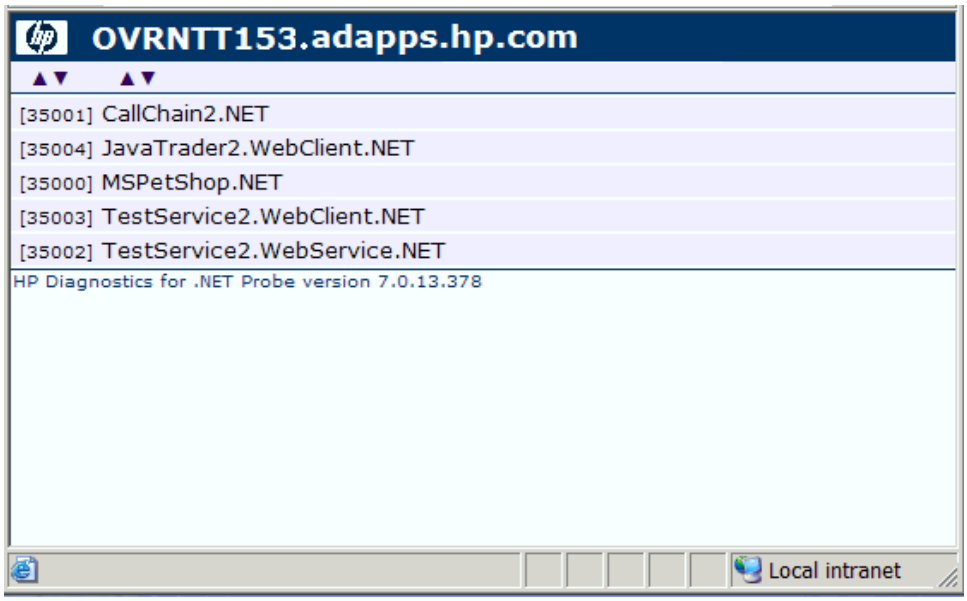
When more than one probe is running on a single host, you cannot know which port each probe is using since the port that is assigned is based on the one that is available at the time the application (and probe) is started. As the applications are started and stopped, the port that is assigned to the probe for a given application is likely to change.

You can determine which probes are running on a host and the ports that they are using by accessing the following URL:

```
http://<probe_host>:<port>
```

For the port value, enter the port number `35000` or `35001`. It does not matter which one you enter.

The list of probes and ports is displayed as shown in the following example:



Authentication and Authorization for .NET Profilers

You can manage the authentication and authorization of users of the Profiler in the **<probe_install_dir>/etc/probe_config.xml** file.

> **Note:** If the .NET Agent is configured to work with a Diagnostics Server, the probe (Profiler) authorization and authentication settings are managed from the Diagnostics Server in Commander mode to which this probe is connected. For more information, see "User Authentication and Authorization" in the HP Diagnostics Server Installation and Administration Guide.
>
> When you access the probe from the Diagnostics Server, the default username is `admin` and the default password is `admin`.

If the .NET Agent is installed as a profiler only, by default, users are not required to enter a username and password to access the profiler.

However, you can configure the profiler to require user authentication. If you configure the profiler to require user authentication, you can define the password required for accessing the profiler.

To configure the profiler to require user authentication:

Go to the **<probe_install_dir>/etc/probe_config.xml** file and set the value of **profiler authenticate** to **true**.

```
<profiler authenticate="true">
  <authentication username="Test" password="uU8X9zOtl6Twi7TkGAhQ="/>
</profiler>>
```

If you do not set a username and password, the default username is `admin` and the default password is `admin`.

To create new usernames and passwords for users of the .NET Diagnostics Profiler:

1. Generate a new username and password using the **PassGen.exe** utility located in the **<probe_install_dir>/bin** directory. Enter the user name and password for encryption. The encrypted password generated for the user is FIPS-2 compliant.

2. In the **probe_install_dir>/etc/probe_config.xml** file, after the **<profiler authenticate="true">** line, enter the username and password for each new user, in the following format:

```
<profiler authenticate="true">
  <authentication username="" password=""/>
</profiler>
```

   - For **authentication username**, enter the username that you chose when running the PassGen utility.

   - for **password**, enter the encoded string that was returned by the **PassGen.exe** utility.

   Caution: If you defined new usernames and passwords to access the profiler, you can no longer use the default username and password (`admin`, `admin`). Rather, you must use one of the new usernames that you defined.

Configuring Consumer IDs

Web service metrics can be grouped by particular consumers of the Web service. The metrics are then aggregated for that consumer and displayed as such in the Services by Consumer ID and Operations by Consumer ID views.

Aggregating the data by consumer ID is useful if you want to determine who is using a particular service and how frequently they are using it. Consumer IDs are also useful for BSM. BSM users can look at the performance of the same application based on consumers to compare their performance characteristics.

Configuring Consumer IDs is optional. By default, the Consumer ID of a Web service being monitored is reported as the IP address of the consumer of the Web service.

There are three ways of defining the consumer ID:

- a value that appears in the SOAP request
- a value that appears in an HTTP header
- to a specific IP address or a range of IP addresses

Basic Procedure for Consumer ID Configuration

The basic procedure to configure consumer IDs is as follows:

1. For each .NET probe for which you want metrics grouped by consumer, update the probe_ config.xml file as described in "Consumer ID Rules Syntax and Examples for .NET Agent" below.

2. If you are configuring more than 5 consumer types, update the **max.tracked.ids.per.probe** setting in the **server.properties** file.

About Consumer ID Rules

The assignment of consumer IDs is controlled by consumer ID rules in the **probe_config.xml** file.

Each category of consumer IDs has its own rules: SOAP rules, HTTP header rules, and IP rules. The rules are applied in an order no matter which order the rules are defined. The SOAP header rules are applied first, the HTTP headers rules are applied next, and lastly the IP rules are applied.

All rule types do not need to be used. There could be SOAP rules, no HTTP rules, and IP rules. If there is no match on any of these rules, the original IP address is used as the consumer ID.

The SOAP rules allow for the consumer ID to be obtained from an XML element in the SOAP header, envelope or body. The rule specifies a regular expression that is used to match against the web service name being called by the consumer. See "Using Regular Expressions" in the HP Diagnostics Server Installation and Administration Guidefor information on using regular expressions.

If there is a match with the web service name, the agent/probe attempts to find the element defined in consumeridfield in the appropriate SOAP location defined by the SOAP rule. If the element is not found, this rule is skipped and the agent/probe goes on to the next rule that is defined.

The HTTP header rules allow for the consumer ID to be obtained from a header in the collection of HTTP headers in a HTTP request.

The IP rules allow for the consumer ID to be obtained from the mapping of IP addresses to a consumer ID. The rule is used to define an IP address, or a range of addresses, to be assigned to a consumer ID.

Consumer ID Rules Syntax and Examples for .NET Agent

The rules syntax and examples are specific to how the consumer ID is being defined.

SOAP Rules

The SOAP rules allow for the consumer ID to be obtained from an XML element in the SOAP header, envelope or body.

An example of configuring consumer ID based on a value in the SOAP header is shown below:

```
<consumeridrules enabled="true">
  <soaprules>
```

```
    <soaprule id="SOAP1" rule="TestService" location="soap-header"
consumeridfield="Caller"/>
  </soaprules>
</consumeridrules>
```

id= attribute can be any name you would like to use to identify the rule; this attribute is not used by the .NET probe.

rule= attribute must be defined for a soaprule. The rule is a regular expression that is used to match against the web service name being called by the consumer or you can use the exact Web service name.

location= can be set to "soap-header", "soap-envelope", "soap-body". If you do not specify a location, it defaults to use "soap-header." If you configure a location for any soap rule, you must configure a location for all soap rules, or a severe error will occur and the consumer ID based on SOAP logic will be disabled.

consumeridfield= attribute must be defined for a soaprule. The element in the SOAP header, envelope or body whose value you want to use as the consumer ID.

If there is a match with the pattern specified in the rule= attribute, the .NET agent attempts to find a text element for the element defined in the consumeridfield. The element in the consumeridfield can be a qualified name—that is, composed of a namespace name and the local part—or an unqualified name, which does not have an associated namespace. If the element is not found in the specified location, this rule is skipped and the probe goes on to the next rule that is defined.

For example, the following rule matches on a Web service named TestService and uses the Caller element's value as the consumer ID:

```
<soaprule id="SOAP1" rule="TestService" location="soap-header"
consumeridfield="Caller"/>
```

As long as the callers of the TestService Web service have a value defined for Caller, the metrics will be grouped by the different values for Caller. Here is an excerpt from the soap header that would map to a consumer ID of "Customer2" for this caller of the TestService:

```
  SoapTest1;WS<env:Envelope
xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <env:Header>
    <Caller>Customer2</Caller> <-- The consumer id returned is"Customer2"
  </env:Header>
  <env:Body env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <m:sell xmlns:m="http://www.bea.com/examples/Trader">
      <string xsi:type="xsd:string">sample string</string>
      <intVal xsi:type="xsd:int">100</intVal>
    </m:sell>
  </env:Body>
</env:Envelope>
```

Enable SOAP Capture

SOAP envelopes can be very large so the <soapcapture> element is provided to enable you to control the overhead, mainly memory overhead, of capturing SOAP requests and responses.

<soapcapture enabled="true">

The <soapcapture> element controls whether SOAP requests and responses are captured. If it is disabled, SOAP requests and responses will not be captured. This means there will not be SOAP requests or responses included in TransactionVision events, nor will there be any SOAP requests available with SOAP faults, and you cannot configure consumer ID based on SOAP header, envelope, or body.

The <soapcapture> setting overrides the settings in <soaprequestforsoapfault> which controls SOAP payload capture on SOAP faults. See "Configuring SOAP Fault Data" on the next page.

HTTP Header Rules

The HTTP header rules allow for the consumer ID to be obtained from a header in the collection of HTTP headers in a HTTP request. A rule and consumeridfield attribute must both be defined for a HTTP rule element, and an id attribute can also be defined for the user to identify individual rules.

The rule is a regular expression that is used to match against the URL that the HTTP request is being sent to by the consumer. If there is a match, the .NET probe attempts to find an HTTP header for the header name defined in the consumeridfield. If the header name is not found in the collection of HTTP headers, this rule is skipped and the probe goes on to the next rule that is defined.

Example httpheader rules:

```
<consumeridrules enabled="true">
  <httpheaderrules>
   <httpheaderrule id="httpHeader 1" rule="/Webservice/.*
consumeridfield="Caller"/>
  </httpheaderrules>
</consumeridrules>
```

IP Address Rules

The IP rules allow for the consumer ID to be obtained from the mapping of IP addresses to a consumer ID. A rule and consumerid attribute must both be defined for an IP rule element, and an id attribute can also be defined for the user to identify individual rules.

The rule is used to define an IP address, or a range of addresses, to be assigned to a consumer ID. This rule can be defined as a single IP address; for example, 19.225.17.125. The rule can also define a range; for example, 19.255.17.125,19.255.17.255.

An asterisk can also be used in an octet of an IP address to match against anything in that octet; for example, 19.255.17.*. A range can be defined in an octet to match a range of values in that octet; for example, 19.255.17.20-255. Combinations of these can also be used; for example, 19.*.17.20-255, 20.*.10-55.*. If there is a match, the .NET probe sets the consumer ID to the consumer ID defined in the rule.

Examples:

```
<consumeridrules enabled="true">
  <iprules>
    <iprule id="IpTest1" rule="18.*.1-20.*" consumerid="Client1"/>
    <iprule id="IpTest2" rule="17.*.*.*" consumerid="Client2"/>
    <iprule id="IpTest3" rule="19.255.17.125,19.255.17.255"
consumerid="Client3"/>
  </iprules>
</consumeridrules>
```

Configuring SOAP Fault Data

If a SOAP fault is detected, the SOAP payload can be included with the SOAP fault data. SOAP payload is only captured when there is a SOAP fault.

In the Diagnostics UI, you can view the payload information as part of the SOAP fault instance tree (call profile).

Because payloads can contain sensitive information such as credit card numbers, payload capture on SOAP faults is disabled by default. To enable payload capture on SOAP faults set **<soaprequestforsoapfault enabled="true"/>** in the **probe_config.xml** file on the .NET probe system.

You can also define the limit for the payload size using the **maxsize** attribute in the **<soaprequestforsoapfault>** element. For example, the following entry increases the SOAP payload length to 10000 from its default of 5000:

```
<soaprequestforsoapfault enabled="true" maxsize="10000"/>
```

The <soapcapture> element overrides the <soaprequestforsoapfault> element. So that if <soapcapture> is disabled, <soaprequestforsoapfault> is disabled even if <soaprequestforsoapfault> is set to true. Also whatever <soapcapture> maxsize value is set, overrides the <soaprequestforsoapfault> maxsize. So that is <soapcapture> maxsize is set to 5000 and <soaprequestforsoapfault> maxsize is set to 10000, the payload size will be maximum of 5000.

Collecting Additional Probe Metrics or Modifying Probe Metrics

You can configure the .NET agent to collect additional probe metrics based on perfmon counters using the <metrics> and <metric> elements in the **<probe_install_dir>\etc\probe_config.xml** file. See "<metric> element" on page 131 and "<metric> element" on page 131 for details.

You can also modify probe metrics using the <metric> element. But note the following special cases:

- If you want to move a metric from one metric category to another, you must change the metric's group attribute as well as the metric name attribute. This is because the existing metric name is

already registered to its old group on the Diagnostics mediator and this association cannot be changed.

- If you want to redefine an existing probe metric it is better to create a completely new metric entry rather than assigning a different perfmon counter to the metric. This ensures that you avoid aggregating disparate data.

Performance Counter Security

The .NET Agent uses Performance Counters to collect probe metrics. This requires the application process that is being monitored by the .NET Agent to have access rights to performance counters. Each process runs as a user account therefore this user account must have access rights to performance counters. The simplest way to do this is to add the user account that the process runs as to the **Performance Monitor Users** group.

However Microsoft has introduced the concept of a **virtual accounts** in Windows Vista SP2, Windows Server 2008 SP2, Windows 7 and Windows Server 2008 R2 (see http://technet.microsoft.com/en-us/library/dd548356(WS.10).aspx for details). These operating systems have used the virtual accounts concept in IIS and by default, application pools in IIS run as **ApplicationPoolIdentity**. Because this user account is virtual, it requires special steps to add the user account to the Performance Monitors Users group.

In Windows 2008 R2 and Windows 7 do the following:

1. Open the **Server Manager** tool, there are many ways to do this but one is through Administrative Tools.
2. In the left hand pane find L**ocal Users and Groups** under Configuration.
3. Click the **+** to expand it.
4. Double-click Groups.
5. Double-click the **Performance Monitor Users** group.
6. Click the **Add…** button.
7. Click the **Locations…** button.
8. Select the local computer.
9. Click the **OK** button.
10. Make sure that object types includes **Built-in security principals**.
11. Enter **IIS APPPOOL\<name of the application pool>**, (example IIS APPPOOL\My WebService App Pool, where My WebService App Pool is the name of the application pool), in the text box.
12. Click the **OK** button.

In Windows 2008 SP2 and Windows Vista SP2 do the following:

1. Open a Command Prompt window.
2. Type **net localgroup "Performance Monitor Users" "IIS APPPOOL\<name of application pool> /ADD** (where <name of the application pool> is the application pool name).
3. **The command completed successfully** will be displayed if this is successful.

Manually Enabling Auto-Discovered ASP.NET Applications and Non ASP.NET Services

When installing a .NET Agent, a utility is automatically run that discovers all the ASP.NET applications configured in IIS and adds them to the **probe_config.xml** file with a status of enabled for monitoring.

You can run another utility at any time after installation that not only automatically discovers all the ASP.NET applications configured in IIS, but also discovers all the services running on the machine that are eligible for monitoring. You can then select the specific applications and services that you want to monitor.

To run this utility:

1. From the **Start** menu, select **All Programs > HP Diagnostics .NET Probe > Run HP .NET AppScanner**.
2. The utility runs and a Window opens with two tabs. The first lists the discovered ASP.NET applications and the second lists the discovered services. Select the relevant tab.
3. Select the check box for each ASP.NET application or service you want to enable for monitoring.

   Note: If an application or service has already been configured and enabled for monitoring, the check box is already selected by default.

4. Click **OK**.

A description of the highlighted application or service appears on the right.

A summary section at the bottom provides additional statistical data.

When you enable a service for monitoring, an element is added to the **probe_config.xml** file with the name of the service's related process. This element includes instrumentation for standard .NET frameworks (ASP.NET, ADO, WCF, EF) as well as a custom instrumentation **.points** file. Configure this file is you require custom instrumentation beyond that which is included in the standard frameworks.

Chapter 8: .NET System Metrics Agent - Systems Metrics Capture

Information is provided about system metrics capture and how to configure the system metrics collector installed with the .NET Agent.

This chapter includes:

- "About the .NET System Metrics Agent" below
- "System Metrics Captured by Default" below
- "Configuring .NET System Metrics Capture" on the next page
- "Adding System Metrics Using the Windows Performance Monitor" on page 205
- "Default Entries in the .NET Agent metrics.config File" on page 207
- "Keywords in the metrics.config File" on page 208

About the .NET System Metrics Agent

A system metrics collector is installed with the .NET Agent and run as a Windows Service (HP Diagnostics Metrics Agent). The .NET system metrics agent gathers system level metrics, such as CPU usage and memory usage, from the agent's host. It is configurable so you can control which metrics are collected as well as aspects of how the metrics are collected and published.

Only one instance of the .NET system metrics agent is run on a given host, no matter how many instances of the probe were started on the host.

> **Note:** To configure additional probe metric capture with the .NET Agent (other than system metrics capture described here) see "Collecting Additional Probe Metrics or Modifying Probe Metrics" on page 199.

System Metrics Captured by Default

The following are the system metrics that the .NET system metrics agent collects by default for all supported platforms (excluding z/OS):

- CPU
- MemoryUsage

- VirtualMemoryUsage
- ContextSwitchesPerSec
- DiskBytesPerSec
- DiskIOPerSec
- NetworkBytesPerSec
- NetworkIOPerSec
- PageInsPerSec
- PageOutsPerSec

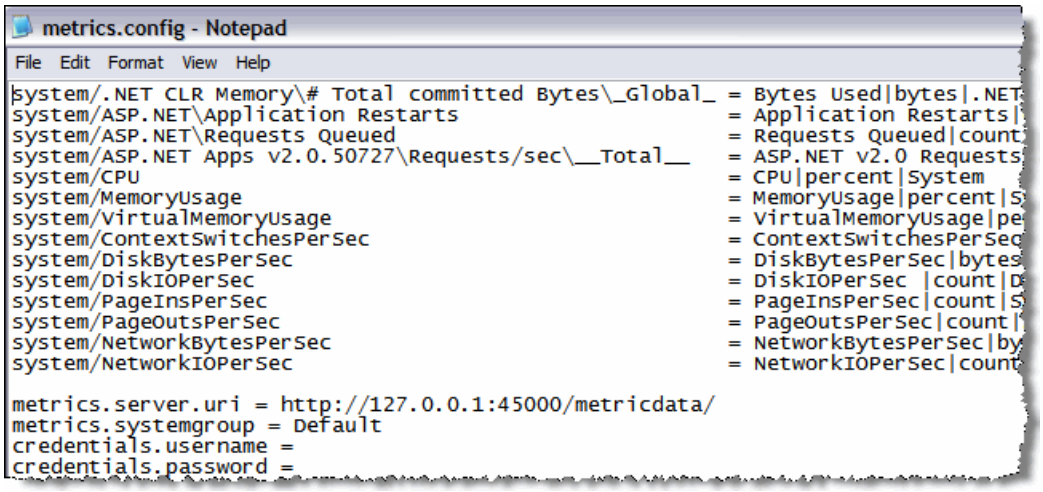
In addition to the default system metrics listed above, the following system metrics are also captured by default on .NET Agent systems. (The layout of these entries is described in "Understanding the system/ Metrics Collector Entries" on the next page).

- .NET CLR Memory\# Total committed Bytes_Global_
- ASP.NET\Application Restarts
- ASP.NET\Requests Queued
- ASP.NET\Request Wait Time
- ASP.NET\Requests Rejected
- ASP.NET Applications\Requests/sec
- ASP.NET Applications\Requests Executing

You can control which of the default system metrics the .NET system metrics agent gathers and you can capture custom system metrics with the .NET system metrics agent.

Configuring .NET System Metrics Capture

The configuration file for the .NET system metrics agent is the **<probe_install_dir>/etc/metrics.config** file. Changes to the metrics.config file are processed dynamically by the .NET Agent.



There is a different **metrics.config** file included with the Java Agent. See the HP Diagnostics Java Agent Guide.

Understanding the system/ Metrics Collector Entries

Metrics collector entries in the **metrics.config** file instruct the .NET system metrics agent to gather specific metrics. Entries that begin with **system/** are processed as Windows Performance Monitor Counters.

These system metrics collector entries use the following layout:

```
system/<Counter_name>\<Performance_object>\<Instance>\<Remote_machine> =
<metric_id>|<metric_units>|<category_id>
```

All fields are required except for the optional <Instance> and <Remote_machine> fields.

Where:

- **Counter_name** indicates the Windows Performance Monitor counter. See "Adding System Metrics Using the Windows Performance Monitor" on the next page for details on how to identify the counter, performance object and instance in the Windows Performance Monitor UI.
- **Performance_object** indicates the Windows Performance Monitor performance object associated with the Counter_name.
- **Instance** indicates the Windows Performance Monitor instance of a counter. You may use a wildcard (*) to indicate that all instances are desired. If you wish to specify a specific enumeration of all instances, you precede the enumeration index number with the hash sign (#1). The enumeration index number must be a positive number.
- **Remote_machine** is only required if the Windows Performance Monitor Counter is running on a machine that is different (remote) from the machine that the .NET system metrics agent is running on. The minimum requirement for this configuration to work is that the Network Service User on the

machine that the .NET system metrics agent is running on must have permissions to read the Windows Performance Monitor Counters from the remote machine.

- **<metric_id>** indicates the name that represents the metric in the Diagnostics UI. The metric_id must be unique in the **metrics.config** file. If the value of the metric_id is the same as one of the default metrics, Diagnostics replaces the metric_id in the entry with a standard name to be used to reference the metric in the UI. If the value of the metric_id is not the same as one of the default metrics, the metric_id is used as the name of the metric in the UI exactly as shown in the entry.
- **<metric_units>** indicates the units of measure in which the metric is reported. This is a required parameter and it must contain one of the following units of measure:
  - microseconds, milliseconds, seconds, minutes, hours, days
  - bytes, kilobytes, megabytes, gigabytes
  - percent, fraction_percent
  - count
  - load
- **<category_id>** groups a set of metrics together under the same heading in the Details pane of the Diagnostics UI. This parameter has no impact on the data displayed in the Diagnostics views.

Example without an <Instance>:

```
system/ASP.NET\Requests Queued = Requests Queued|count|ASP
```

Example with an <Instance>:

```
system/Processor\% Processor Time\_Total = CPU|percent|System
```

Example with an integer <Instance>:

```
system/Processor\% Processor Time\#1 = CPU 1|percent|System
```

Example without an <Instance> and running on a <Remote_Machine):

```
system/ASP.NET\Requests Queued\\IISAQUAH = Requests Queued(IISAQUAH)|count|ASP
```

Adding System Metrics Using the Windows Performance Monitor

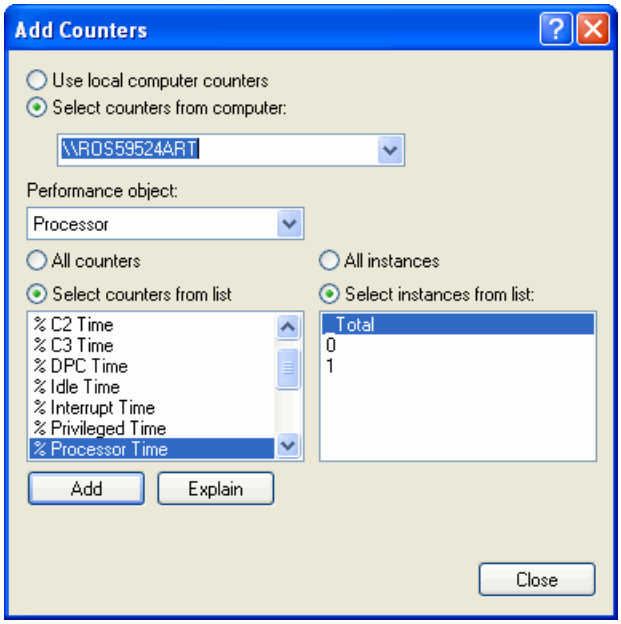
To add a system metric counter to the metrics.config file you must first find its definition using the Windows Performance Monitor (Perfmon). The following example uses version 5.x of Perfmon. Version 6.x is similar but the UI is a little different.

To add counters in Perfmon:

1. Start the Windows Performance Monitor. For example select **Start > Control Panel > Administrative Tools > Performance**.

2. The Perfmon Performance dialog box is displayed showing the System Monitor graph with a table of the current counters beneath the graph. Right-click the System Monitor graph and select **Add Counters...** from the pop-up menu.

   The Add Counters dialog box is displayed:



3. Select the **Select counters from computer** entry and make sure the host computer is select in from the drop down list.

4. In the Performance object list, select the object that the counter belongs to.

5. Choose **Select counters from list** and select an instance from the list of instances.

6. Click the **Add** button to add the counter. The following instructions tell you how to create an entry for a counter using the system/ metrics entry described in "Understanding the system/ Metrics Collector Entries" on page 204.

To collect metrics for a Perfmon counter:

1. Open the **<probe_install_dir>/etc/metrics.config** file on the .NET agent system.

2. Create the **system/** metrics entry for the counter using the layout described in "Understanding the system/ Metrics Collector Entries" on page 204.

   You can add this entry anywhere in the file, however best practice is to add it to the bottom of existing collection of these type of entries. In the example shown in the screen shot above:

- The selected host computer is ROS59524ART
- The selected Performance object is Processor
- The selected Counter is % Processor Time
- The selected Instance is _Total

So if the host computer is local, the entry in the metrics.config file for the Performance Monitor counter would be:

```
system/Processor\% Processor Time\_Total = CPU|percent|System
```

And if the host computer is remote, the entry in the metrics.config file for the Performance Monitor counter would be:

```
system/Processor\% Processor Time\_Total\ROS59524ART = CPU(ROS59524ART)
|percent|System
```

Performance Counter Security

The .NET metrics agent uses Performance Counters to collect system metrics. The metrics agent runs as a Network Service and this account needs to be added to the **Performance Monitor Users** group.

Troubleshooting Added System Metrics Counters

If you specify a new counter that appears to not be functioning, you can use the Windows Event Viewer to look at the Diagnostics logs for the .NET system metrics agent source for errors and warnings.

For example:

A **Could not locate Performance Counter with specified category name** warning entry typically indicates that you may have mis-typed the name of the counter. This can happen, for example, if you read a counter name from the PerfMon Performance pane that has embedded blanks. The default font used by PerfMon is not a monospaced font and as such makes it difficult to see embedded blanks in the name of the counters, categories and instances. You can change the font to a monospaced font type and then more clearly see the exact format of counter names.

For example:

An **Instance does not exist in the specified Category** warning entry typically indicates that the instance you have chosen is not active at this time. We do not recommend that you use transient instances. Permanent instances like __Total__ are appropriate.

Default Entries in the .NET Agent metrics.config File

Upon installation, the **<probe_install_dir>/etc/metrics.config** file has three entries:

- A grouping of default **system/** entries for PerfMon counters
- A **metrics.server.uri** entry that specifies how the .NET system metrics agent publishes its data
- A default **metrics.systemgroup** entry

Other additional entries can be added after these default entries.

Keywords in the metrics.config File

The keywords that can be used in entries in the **<probe_install_dir>/etc/metrics.config** file are as follows:

- credentials.password
- credentials.username
- default.sampling.rate
- metrics.server.uri
- metrics.systemgroup
- metrics.agent.publish.interval
- metrics.agent.registered_hostname
- proxy.password
- proxy.user
- proxy.uri
- system/

The use of the **system/** keyword is described in "Configuring .NET System Metrics Capture" on page 203.

The use of each of the other keywords is described in the following section.

credentials.password	This setting must match the setting for the password attribute of the <credentials> element in the probe_config.xml file. See "<credentials> element" on page 96 for more details.
credentials.username	This setting must match the setting for the username attribute of the <credentials> element in the probe_config.xml file. See "<credentials> element" on page 96 for more details.

default.sampling.rate	This setting defines the rate at which the .NET system metrics agent samples the configured system metric counters. The default rate is every 5 seconds. Values are expressed as a number of Seconds, Minutes, Hours or Days, for example, nS, nM, nH or nD. The following example sets the rate to every 10 seconds: default.sampling.rate = 10s

metrics.server.uri	This setting is automatically generated at install time. It defines the URI that the .NET system metrics agent uses to publish the system metric counters to the Diagnostic Mediator Server.
	
	The following example is for a Diagnostic Mediator Server running on the my_diag_server machine, and using a metricport of 2006 to publish the metrics:
	
	`metrics.server.uri = http://<my_diag_server>:2006/metricdata/?sleep=false`
	
	Any changes to the probe_config.xml settings for either the metrichost attribute or the metricport attribute of the <mediator> element must also be reflected at the same time in the metrics.server.uri setting.
	
	The ?sleep setting controls whether the Diagnostic Mediator Server that receives the published metrics will respond immediately or delay its response to the .NET system metrics agent. A setting of ?sleep=false responds immediately, a setting of ?sleep=true delays its responds by a default of 5 seconds.
	
	The following example is for a Probe Aggregator-enabled .NET system, using the default metricport of 45000 to publish the metrics:
	
	`metrics.server.uri = http://127.0.0.1:45000/metricdata/`

metrics.systemgroup	This setting is automatically generated at install time. Do not change this setting.
metrics.agent.publish.interval	This setting defines the interval between publishes of the current values of the System Metric Counters by the .NET system metrics agent to the Diagnostic Mediator Server. The default interval is 5 seconds. Set values can be expressed as a number of Seconds or Minutes, for example, nS or nM. The following example sets the publish interval to 10 seconds: metrics.agent.publish.interval = 10S

metrics.agent.registered_hostname	Refer to the "Overriding the Default Probe Host Machine Name" on page 192 for a description of when and how to use this setting.
proxy.password	This setting must match the setting for the proxypassword attribute of the < diagnosticsserver> element in the probe_config.xml file. See "<diagnosticsserver> element" on page 99 for more details. Also refer to “Configuring Diagnostics Servers and Agents for HTTP Proxy” in the HP Diagnostics Server Installation and Administration Guide.
proxy.user	This setting must match the setting for the proxyuser attribute of the < diagnosticsserver> element in the probe_config.xml file. See "<diagnosticsserver> element" on page 99 for more details. Also refer to “Configuring Diagnostics Servers and Agents for HTTP Proxy” in the HP Diagnostics Server Installation and Administration Guide.
proxy.uri	This setting must match the setting for the proxy attribute of the < diagnosticsserver> element in the probe_config.xml file. See "<diagnosticsserver> element" on page 99 for more details. Also refer to “Configuring Diagnostics Servers and Agents for HTTP Proxy” in the HP Diagnostics Server Installation and Administration Guide.

Part 4: Using the Profiler for .NET

Chapter 9: Diagnostics Profiler for .NET

This chapter describes how to use the .NET Diagnostics Profiler:

- " About the .NET Diagnostics Profiler" below
- "How the .NET Agent Provides Data for the .NET Profiler" on the next page
- ".NET Diagnostics Profiler UI Navigation and Display Controls" on page 214
- ".NET Diagnostics Profiler Inactivity Timeout" on page 215
- "How to Access the .NET Diagnostics Profiler" on page 215
- "How to Enable and Disable the .NET Diagnostics Profiler" on page 216

.NET Diagnostics Profiler UI Description:

- "Server Requests Tab Description" on page 217
- "SQL Tab Description" on page 220
- "Methods Tab Description" on page 223
- "Call Tree Tab Description" on page 225
- "Exceptions Tab Description" on page 228
- "Collections Tab Description" on page 230

.NET Threads Window UI Description:

- "Threads Window Description" on page 233

About the .NET Diagnostics Profiler

The Diagnostics Profiler for .NET is installed with the .NET Agent. The Profiler runs in a separate UI and provides near real-time data, enabling you to pinpoint application performance bottlenecks.

> **Note:** The.NET Diagnostics Profiler operates in an unlicensed mode with load restrictions until the probe is able to connect to a Diagnostics Server that has been properly licensed. In unlicensed mode, the .NET Profiler is limited to capturing data from 5 concurrent threads.
>
> If you installed the unlicensed trial software agent from the HP Software Web site and you want to use it with a Diagnostics Server, contact HP Software Support to purchase HP Diagnostics.

If you are using Diagnostics with HP LoadRunner or HP Performance Center you will be prompted to enter the Diagnostics User Name and Password when selecting the .NET Profiler from the Diagnostics UI.

You can use the different tabs in the .NET Profiler to analyze method latency for the selected application. And you can analyze memory problems for the selected application using the memory diagnostics metrics displayed in the .NET Profiler.

Some of the information presented in the .NET Profiler is also available in the Diagnostics enterprise UI.

How the .NET Agent Provides Data for the .NET Profiler

This section describes the way in which the .NET Agent monitor your application and how this data is displayed in the .NET Diagnostics Profiler.

Monitoring Method Latency and Call Stacks

The .NET Agent runs probes to monitor your application and keep track of the metrics for all of the instrumented methods that your application calls. As probes are monitoring, they capture the call stack for the three slowest instances and the single fastest instance of each server request.

When a new server request instance is encountered that is slower than one of the currently captured instances for the server request, it replaces one of the previously captured instances. In the same manner the captured call stack for the fastest instance is replaced when an instance that is even faster is encountered.

The .NET Diagnostics Profiler displays metrics for all of the instrumented methods. The .NET Profiler ignores all configured trim settings, for example, latency trimming, depth trimming or throttling. For details about trim configuration refer to the "Advanced .NET Agent Configuration and Instrumentation" on page 58. You can drill down to the instances of the methods that were included in one of the four server request call stacks that were captured when you accessed the .NET Diagnostics Profiler user interface.

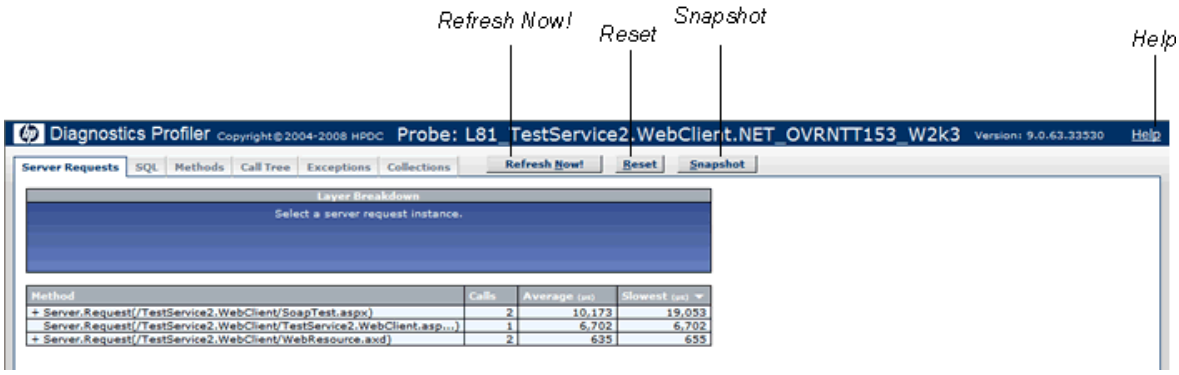
While you are analyzing the information displayed on the various tabs of the .NET Diagnostics Profiler, you are working with the methods and call stacks captured from the time that the .NET Profiler was started/reset to the time that the user interface was started/refreshed. In the meantime the probe continues to monitor your application, capture method metrics, and capture call stacks. These changes are not sent automatically to the user interface, you must request them via the **Refresh Now** button. This is so the underlying data will not change unexpectedly while you are investigating something of interest.

Monitoring Application Memory

The .NET Diagnostics Profiler allows you to monitor your application's memory usage using Lightweight Memory Diagnostics. Lightweight Memory Diagnostics allows you to monitor the collections that your application has created, and to identify the largest collections and the fastest growing collections. For more information about Lightweight Memory Diagnostics, see "Collections Tab Description" on page 230.

.NET Diagnostics Profiler UI Navigation and Display Controls

This section describes the following features and controls that are common to all of the .NET Profiler tabs: **Refresh now**, **Reset**, **Snapshot** and **Help**:



Refresh Metrics

Click **Refresh Now** to refresh the information displayed on the tabs with the latest metrics and call stacks.

After you refresh the metrics, the .NET Diagnostics Profiler continues to monitor and collect metrics using the same baseline for the calculations of instance counts, average latency, and slowest latency. It also continues to use the captured call stacks as a basis of comparison for finding new call stacks to capture.

Reset Metrics

You can force the .NET Diagnostics Profiler to use new baselines for the calculation of instance counts, average latency, and slowest latency, and to force-drop all captured call stacks, by clicking **Reset**.

After you reset the metrics, the .NET Diagnostics Profiler begins collecting data with new baselines and starts processing the instance trees as though the profiler had just been started.

> **Note:** You may want to click **Reset** once your system has warmed up so that you can do your performance analysis using metrics that are more representative of the processing that takes place when your application is running in steady state.

Take a Snapshot

You can capture a snapshot of the data from your profiler session into an .xml formatted file, by clicking the **Snapshot** button.

The resulting snapshot can be used, for example, as a report that is distributed to your colleagues or as a point of reference when you are about to make changes to your applications. The snapshot includes the profiler tabs so that you can review and analyze the data in the snapshot in the same way that you would view it in the Profiler.

The Profiler displays a dialog box that indicates the path to where the .xml file is stored. When you open the snapshot, the saved profiler data is displayed in your browser.

Access Help

When you click **Help**, on the top right hand corner of the screen, you access the on-line help manual for the .NET Diagnostics Profiler.

.NET Diagnostics Profiler Inactivity Timeout

By default, the .NET Diagnostics Profiler is not started until you display the Profiler UI. When you close the Profiler UI, the profiler continues to run for a period of time specified by the **inactivitytimeout** attribute in **<probe_install_dir>/etc/probe_config.xml**. If you reopen the Profiler UI before the profiler times out, the profiler displays the data for the time period since the profiler was started. If you reopen the Profiler UI after the timeout has occurred, the profiler is restarted and only the data for the new profiler session is displayed. As long as the Profiler UI is open, the profiler session remains active. The count down for the inactivity timeout begins when you close the Profiler UI.

How to Access the .NET Diagnostics Profiler

Once you have installed the .NET Agent, configured a probe to collect performance metrics and started the application that is being monitored, you can access the .NET Diagnostics Profiler from your browser and view diagnostics data. You can also access the .NET Diagnostics Profiler by drilling down from the views of the Diagnostics Enterprise user interface.

Remote access to the .NET Profiler can be disabled with the profiler element in the **probe_config.xml** file.

To open the .NET Diagnostics Profiler directly (standalone):

1. In your browser, go to the .NET Diagnostics Profiler URL: http://<probe_host> :< probeport> /profiler

   The probes are assigned to the first available port beginning at **35000**.

2. Type your username and password.

   Depending on your authentication settings, you may be prompted to enter a username and password. The default username is **admin**. The default password is **admin**.

   For more information about authentication, usernames and passwords when you have the full Diagnostics product, refer to the HP Diagnostics Server Installation and Administration Guide section on Authentication and Authorization.

To drill down to the Diagnostics .NET Profiler from the main Diagnostics UI:

1. From any view in the Diagnostics UI that shows probe entities, right-click the probe in the table and select **View Profiler for <probe name>** from the menu.

   If you are using Diagnostics with HP LoadRunner or HP Performance Center you will be prompted to enter the Diagnostics User Name and Password when selecting the .NET Profiler from the Diagnostics UI.

2. If the Profiler fails to open, ensure that you have set a default browser within your operating system.

How to Enable and Disable the .NET Diagnostics Profiler

This task describes how to disable and re-enable the .NET Profiler to start.

When the .NET Agent is installed and probes configured to work with a Diagnostics Server, the probe data collection starts automatically when a Web page in the monitored application is accessed.

By default the .NET Diagnostics Profiler isn't started until you access the Profiler UI. You may configure the agent so that the .NET Profiler is started at the same time that the probe data collection is started or so the .NET Profiler cannot be started.

To configure the probe to automatically start the profiler:

You may want to start the .NET Profiler at the same time that the probe is started if you are trying to understand the performance of your application when it is first invoked.

Set the **pro** attribute in **<probe_install_dir>/etc/probe_config.xml** to `true`.

```
<modes enterprise="true" pro="true"/>
```

To configure the probe to prevent the Profiler from starting:

You may want to prevent someone from starting the .NET Profiler for a probe that is monitoring an application where you do not want to incur the additional overhead from the .NET Profiler.

Set the **pro** attribute in **<probe_install_dir>/etc/probe_config.xml** to `false`

```
<modes enterprise="true" pro="false"/>
```

To configure the probe to start the Profiler when you access the UI:

By default, the probe starts the Profiler when you bring up the Profiler UI. If you have altered the setting for the probe, you may want to reset the behavior of the probe to the default behavior.

Set the **pro** attribute in **<probe_install_dir>/etc/probe_config.xml** to `auto`:

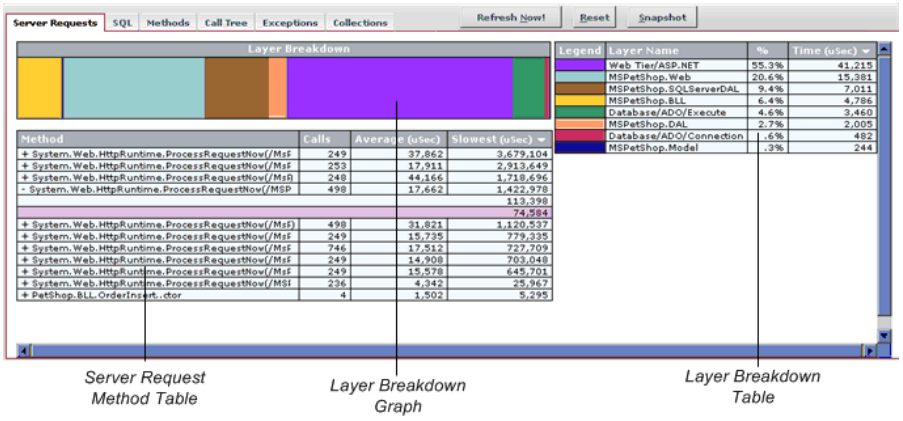
```
<modes enterprise="true" pro="auto"/>
```

Note: If you do not include the **pro** attribute, the probe defaults to the behavior when **pro** is set to `auto`.

Server Requests Tab Description

The .NET Diagnostics Profiler keeps track of all of the method calls made by your application. The Server Requests tab displays information about the server request methods. The server request methods are listed in a table that shows the number of times that each method was executed, along with the average latency and the slowest execution time for all of the calls to the method. You can expand each server request listed in the table, to reveal the latency for the three slowest instances of the server request along with the single fastest instance.

Note: The .NET Diagnostics Profiler captures call trees for the three slowest instances and the single fastest instance of each server request. The .NET Diagnostics Profiler lets you drill into the captured call trees from the Server Requests tab.

UI example	
To access	In the .NET Diagnostics Profiler, select the Server Requests tab.
Relevant tasks	"How to Access the .NET Diagnostics Profiler" on page 215

The following user interface elements are included:

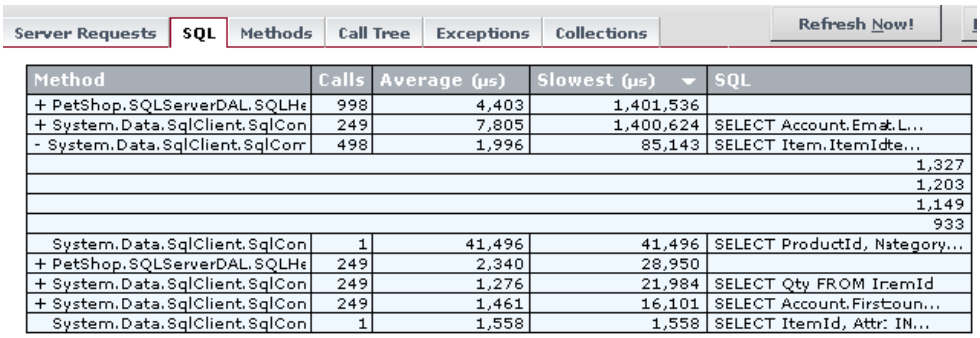
UI Element	Description
Server Request Method Table	The Method table lists the server requests that have been called. You can sort the table by clicking the column headers. The following columns are included in the table: **Method.** The server request methods that were called. If a server request method was called more than once, the method name is preceded by a plus sign (+) or a minus sign (-) to indicate that the instance specific latency information is available for the server request. **Calls.** The number of times that the server request method was invoked. **Average.** The average latency for all of the calls to the server request method. The average latency is shown in microseconds. **Slowest.** The response time of the instance with the longest latency. The slowest response time is shown in microseconds. If a server request method was called more than once, the method name is preceded by a plus sign (+) or a minus sign (-). When you click the plus sign, the entry is expanded to reveal the three slowest instances of the method along with the single fastest method. Click the minus sign to the collapse instances shown. If a server request method was called only once, the entry itself represents the single instance of the method call. The value in the Slowest column is the instance's latency. You can view the call tree for a server request instance by clicking on any row that contains a server request instance (a row that does not have a plus sign (+) or a minus (-) sign before the method name or that only contains a latency value is a server request instance). The Profiler switches to the Call Tree tab and displays the call tree for the selected server request instance. The method call for the selected server request is highlighted in blue in the call tree.

UI Element	Description
Layer Breakdown Graph	The Layer Breakdown graph shows the amount of processing time that was spent in each layer while executing a selected instance of a method call. It is a graphical representation of the information shown in the Layer Breakdown table. You can view the Layer Breakdown for a server request instance by hovering the mouse pointer on any row in the Method table that contains a server request instance (a row that does not have a plus sign (+) or a minus (-) sign before the method name, or that only has a latency value, is a server request instance). The Profiler shows the layer breakdown for the indicated instance in both the Layer Breakdown Graph and Layer Breakdown Table. The graph is divided so that each layer is depicted as an area on the graph that is proportional to the percentage of processing that was performed in the layer. Each layer is displayed in a different color, as shown in the Legend column in the Layer Breakdown table.
Layer Breakdown Legend	The Legend shows the amount of processing time that was spent in each layer while executing a selected instance of a method call. The table can be sorted by clicking the column headers. The following columns are included in the table: **Legend.** The color that is used in the Layer Breakdown graph to depict the processing that took place in the layer. **Layer Name.** The name of the layer where the processing for the server request took place. **%.** The percentage of processing time that was spent in each layer, for a selected server request. **Time.** The latency measured for the processing that took place in the layer, for a selected server request. The time is shown in microseconds.

SQL Tab Description

The .NET Diagnostics Profiler keeps track of all of the method calls that your application makes. The SQL tab displays the SQL methods only. The SQL methods are listed in the Method table which shows the number of times that each method was executed, along with the average latency and the slowest execution time for all of the calls to the method. The Method table also shows the actual SQL statement when it was included in the SQL method call.

Each SQL method listed in the table can be expanded to reveal the latency for each instance of the method that was included in a captured call tree.

UI example	 Server Requests | SQL | Methods | Call Tree | Exceptions | Collections | Refresh Now!
To access	In the .NET Diagnostics Profiler, select the SQL tab.
Important information	The .NET Diagnostics Profiler captures call trees for the three slowest instances and the single fastest instance of each server request. You can drill down to the captured call trees from the SQL tab.
Relevant tasks	"How to Access the .NET Diagnostics Profiler" on page 215
See also	For more information on the Call Tree tab, see "Call Tree Tab Description" on page 225.

Method	Calls	Average (µs)	Slowest (µs)	SQL
+ PetShop.SQLServerDAL.SQLHe	998	4,403	1,401,536	
+ System.Data.SqlClient.SqlCon	249	7,805	1,400,624	SELECT Account.Email.L...
- System.Data.SqlClient.SqlCon	498	1,996	85,143	SELECT Item.ItemIdte...
				1,327
				1,203
				1,149
				933
System.Data.SqlClient.SqlCon	1	41,496	41,496	SELECT ProductId, Category...
+ PetShop.SQLServerDAL.SQLHe	249	2,340	28,950	
+ System.Data.SqlClient.SqlCon	249	1,276	21,984	SELECT Qty FROM ItemId
+ System.Data.SqlClient.SqlCon	249	1,461	16,101	SELECT Account.FirstCoun...
System.Data.SqlClient.SqlCon	1	1,558	1,558	SELECT ItemId, Attr: IN...

The following user interface elements are included:

UI Element	Description
Table	This table lists the SQL methods that have been called, and displays latency information for instances of the SQL method calls that were included in the captured call trees. The table can be sorted by clicking the column headers. The following columns are included in the table: **Method.** The SQL methods that were called. If an SQL method has two or more instances in the captured call trees, the method name is preceded by a plus sign (+) or a minus sign (-) to indicate additional instance specific latency information can be viewed for the SQL call. **Calls.** The number of times that the SQL method was invoked. This count includes all instances, whether or not they are included in the captured call trees. **Average.** The average latency for all of the calls to the SQL method. The average latency is shown in microseconds. **Slowest.** The response time for the instance with the longest latency. The slowest response time is shown in microseconds. **SQL.** The first part of the SQL statement that was executed by the SQL method call. You can display a tooltip containing the entire SQL statement by holding the mouse pointer over a row in the SQL column. The latencies for instances of SQL methods can be displayed if they are included in one of the captured call trees. If two or more instances of an SQL method are included in the captured call trees, that method's name is preceded by a plus sign (+) or a minus sign (-) in the Method table. The entry can be expanded to reveal the latency for each of the captured instances for the selected method. Click the minus sign to collapse the visible instances.

UI Element	Description
Table (continued)	if only one instance of an SQL method was included in the captured call trees, the method name in the SQL Method table is not preceded by a plus sign or minus sign and the table entry itself represents the single instance of the method call, and the value in the Slowest column is the instance's latency. If no instances of a SQL method were included in the captured call trees, the method is not preceded by a plus sign or minus sign, and when you click the method, you get a message indicating that although this method was called there is no data captured for it. You can view the call tree for an SQL method instance listed in the SQL Method table by clicking on any row that contains an instance of an SQL method call. (A row that does not have a plus sign (+) or a minus (-) sign before the method name, or that only contains a latency value, is an SQL instance.) When you select a row with an SQL method instance, the Call Tree tab opens, and displays the call tree for the selected SQL method instance. The method call for the selected SQL method is highlighted in blue in the call tree. For information on the Call Tree tab, see "Call Tree Tab Description" on page 225.

Methods Tab Description

The .NET Diagnostics Profiler keeps track of all of the method calls that your application makes. The **Methods** tab is used to list all of the methods. The methods are listed in the Method table, which shows the number of times each method was executed, along with the average latency and the slowest execution time for all of the calls to the method. The methods listed in the Methods tab include the server requests methods listed in the Server Requests tab, the SQL methods listed in the SQL tab, and the methods that generated exceptions shown in the Exceptions tab.

Each method listed in the table can be expanded to reveal the latency for each instance of the method that was included in one of the captured call trees. The .NET Diagnostics Profiler captures call trees for the three slowest instances and the single fastest instance of each server request. The .NET Diagnostics Profiler lets you drill down to the captured call trees from the Methods tab.

UI example	Server Requests | SQL | Methods | Call Tree | Exceptions | Collections — Refresh Now! — Reset Method / Calls / Average (µs) / Slowest (µs) + System.Web.HttpRuntime.ProcessRequestNow — 3475 — 21,627 — 3,679,104 + PetShop.Web.Global.Application_Error — 248 — 14,396 — 1,701,324 + PetShop.Web.OrderProcess.OnLoad — 248 — 23,314 — 1,556,395 + PetShop.Web.ProcessFlow.CartController.PurchaseCart — 248 — 23,061 — 1,550,664 + PetShop.Web.SignIn.SubmitClicked — 249 — 11,543 — 1,405,137 + PetShop.Web.ProcessFlow.AccountController.ProcessLogin — 249 — 10,951 — 1,404,444 + PetShop.BLL.Account.SignIn — 249 — 10,094 — 1,403,582 + PetShop.SQLServerDAL.Account.SignIn — 249 — 9,545 — 1,403,022 + PetShop.SQLServerDAL.SQLHelper.ExecuteReader — 998 — 4,403 — 1,401,536 + System.Data.SqlClient.SqlCommand.ExecuteReader — 998 — 3,351 — 1,400,624 + PetShop.BLL.Cart.GetOrderLineItems — 248 — 3,466 — 720,331
To access	In the .NET Diagnostics Profiler, select the Methods tab.
Relevant tasks	"How to Access the .NET Diagnostics Profiler" on page 215

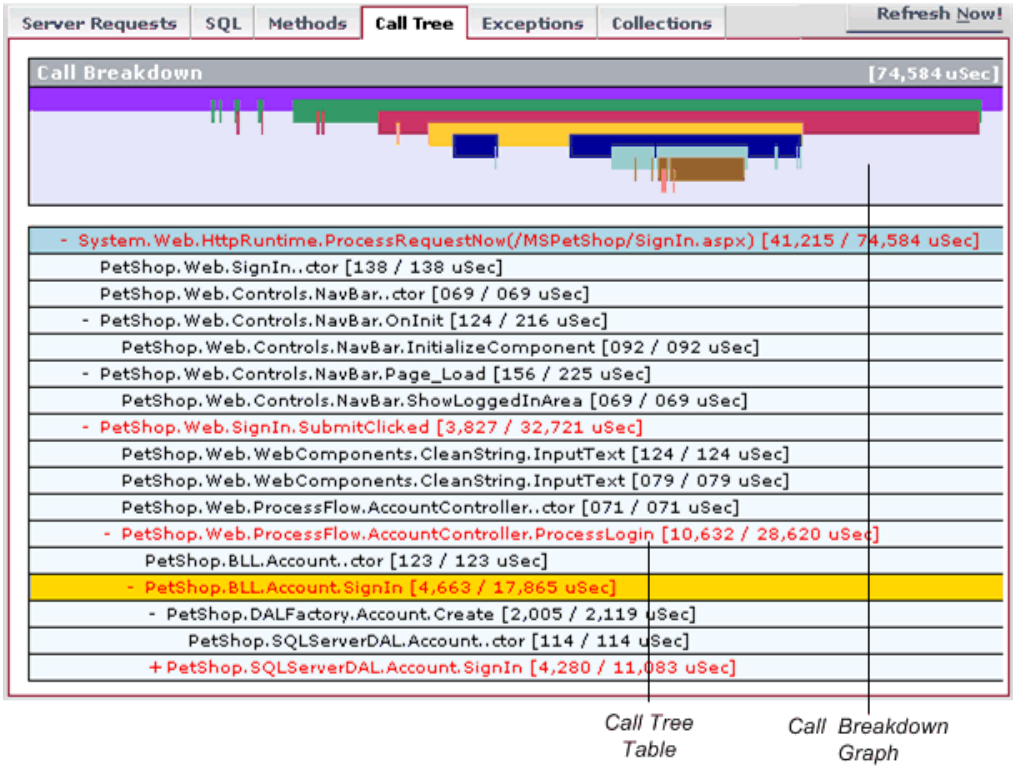
The following user interface elements are included:

UI Element	Description
Table	This table lists the methods that have been called, and displays latency information for instances of the method calls that are included in the captured call trees. This table can be sorted by clicking the column headers. The following columns are included in the table: **Method.** The name of the methods that were called. If a method has two or more instances included in the captured call trees, the method name is preceded by a plus sign (+) to indicate additional instance specific latency information can be viewed for the method call. **Calls.** The number of times that the method was invoked. This count includes all instances, whether or not they are included in the captured call trees. **Average.** The average latency for all of the calls to the method. The average latency is shown in microseconds. **Slowest.** The response time for the instance with the longest latency. The slowest response time is shown in microseconds. You can view the latency for instances of methods if they are included in one of the captured call trees. If two or more instances of a method are included in the captured call trees, the method name in the Method table is preceded by a plus sign (+) or a minus sign (-). The plus sign indicates that you can expand the entry to reveal the latency for each of the captured instances for the selected method. Click the minus sign to collapse the visible instances. If no instances of a method were included in the captured call trees, the method is not preceded by a plus sign or minus sign, and when you click the method, you get a message indicating that although this method was called there is no data captured for it.
Table (continued)	You can view the call tree for a method instance listed in the Method table by clicking on any row that contains an instance of a method call. (A row that does not have a plus sign (+) or a minus (-) sign before the method name, or that only contains a latency value, is a method instance.) When you click a row with a method instance, the Call Tree tab opens and displays the call tree for the selected method instance. The method call for the selected method is highlighted in blue in the call tree. For information on the Call Tree tab, see "Call Tree Tab Description" on the next page.

Call Tree Tab Description

The .NET Diagnostics Profiler captures call trees for the three slowest instances and the single fastest instance of each server request. The captured server request call trees are displayed on the Call Tree tab, in the Call Breakdown graph and in the Call Tree table.

As you analyze the methods presented on the Server Requests, SQL, Exceptions, and Methods tabs, you navigate to the Call Tree tab to understand the context of the processing associated with particular instances of the method's execution. The call tree allows you to see the calling and the callee methods for the method of interest as well as the contribution of those methods to the measured latency.

UI example	
To access	In the .NET Diagnostics Profiler, select the Call Tree tab. You can also access a Call Tree by clicking one of the method instances listed on the Server Requests, SQL, Exceptions, and Methods tabs.
Relevant tasks	"How to Access the .NET Diagnostics Profiler" on page 215

The following user interface elements are included:

UI Element	Description
The Call Breakdown Graph	The Call Breakdown graph shows the processing time that was spent at each level of the call tree hierarchy. Each level in the graph represents the processing at the corresponding level in the call stack. The length of the bar is proportional to the length of time spent in performing the methods at that level of the call stack. The positions where a bar starts and stops indicates the relative time, in relationship to the other levels, that the processing for the level began and ended. A gap in a bar, where the bar ends and then resumes again, indicates that the processing returned to a higher level in the hierarchy before once again proceeding at the lower level. There are two ways that you may identify the method associated with a particular location on the Call Breakdown graph as you mouse over the bars in the graph. As you slide the pointer along a bar in the graph, a tooltip is displayed with the name of the method associated with each segment of the graph bar. As you slide the pointer along a bar in the graph, the Call Tree table scrolls so that the method associated with the selected location in the graph is displayed in the table. The row that contains the selected method is highlighted in gold.

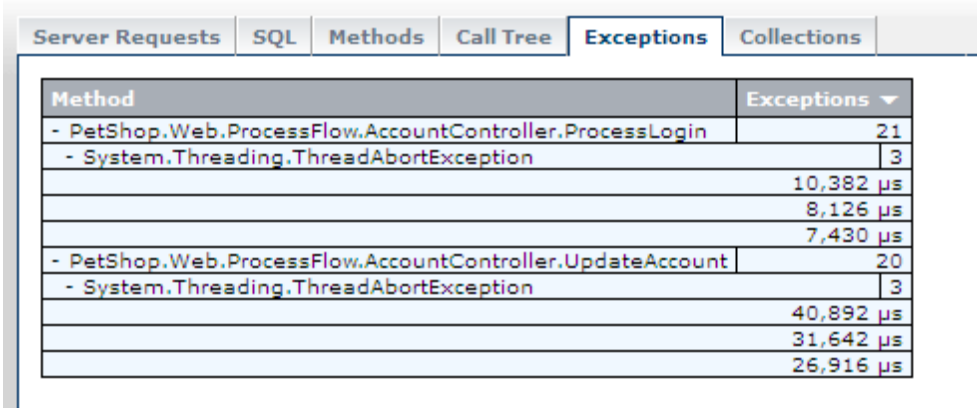
UI Element	Description
Call Tree Table	The Call Tree table lists method calls that are part of a captured server request call tree in a hierarchical structure. Each method in the call tree is depicted on a separate line containing two parts: the method name and the latency. The latency for each method is shown in brackets following the method name. There are two numbers in the brackets separated by a slash: the exclusive latency and the total latency. Exclusive Latency is the amount of latency that is attributable to just the processing in the selected method. Total Latency is the amount of latency that is attributable to the selected method and all of its callee methods. In the following example the exclusive latency is 156: - PetShop.Web.Controls.NavBar.PageLoad [156/225 uSec] To see a captured call tree on the Call Tree tab you must select a method instance from one of the other .NET Diagnostics Profiler tabs. The Call Tree tab opens with the call tree that contains the selected instance visible and the selected method instance highlighted in blue. The method of interest will remain highlighted until a different method is selected on one of the other tabs. You may identify the method associated with a particular location on the Call Breakdown graph by mousing over the bars in the graph. As you slide the pointer along a bar in the graph, the Call Tree table scrolls so that the method associated with the selected location in the graph is displayed in the table. The row that contains the selected method is highlighted in gold. The path through the call tree that has the longest latency is called the critical path. Methods in the Call Tree table that are on the critical path are written using a red font.

Exceptions Tab Description

The .NET Diagnostics Profiler keeps track of all of the method calls that your application makes. The Exceptions tab is used to list only the methods that generated exceptions. The calling methods that generated exceptions are listed in a table that shows the number of times that each method threw an exception. This information allows you to quickly determine if your application is throwing exceptions, and exactly what those exceptions are.

If the exception was included in one of the captured call trees, the exception class will also be listed in the table along with the latency for each instance of an exception.

Note: The .NET Diagnostics Profiler captures call trees for the three slowest instances and the single fastest instance of each server request. You can drill down to the captured call trees from the Exceptions tab.

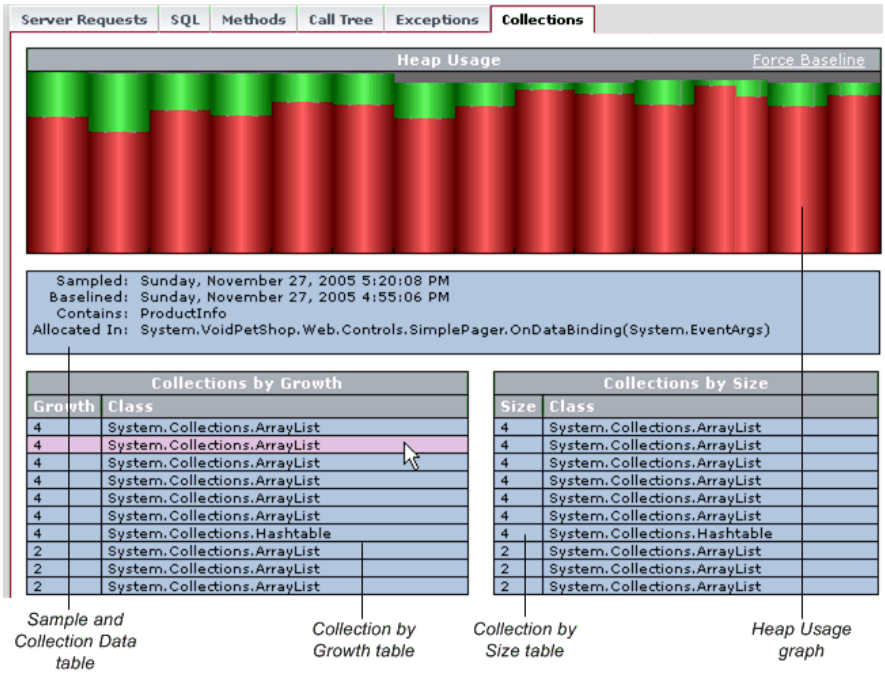
UI example	 Server Requests | SQL | Methods | Call Tree | Exceptions | Collections Method / Exceptions - PetShop.Web.ProcessFlow.AccountController.ProcessLogin 21 - System.Threading.ThreadAbortException 3 10,382 µs 8,126 µs 7,430 µs - PetShop.Web.ProcessFlow.AccountController.UpdateAccount 20 - System.Threading.ThreadAbortException 3 40,892 µs 31,642 µs 26,916 µs
To access	In the .NET Diagnostics Profiler, select the Exceptions tab.
Important information	Exceptions are only captured by the probe if the exception causes the termination of a method. If the instrumented method handles the exception, no exception information is gathered by the probe.
Relevant tasks	"How to Access the .NET Diagnostics Profiler" on page 215

The following user interface elements are included:

UI Element	Description
Table	This table lists the methods calls that generated exceptions and allows you to view latency information for instances of the exceptions that were included in the captured call trees. The rows in this table can be sorted by clicking the column headers. The table includes the following columns: **Method.** The name of the methods generated exceptions. If a method generated two or more exceptions and they were included in the captured call trees, the method name is preceded by a plus sign (+) or a minus sign (-) to indicate that additional instance-specific latency information can be viewed for the exception. **Exceptions.** The number of times that the method generated an exception. This count includes all instances of all classes of exceptions, whether or not they are included in the captured call trees. The latency for instances of exceptions are available to be displayed if they are included in one of the captured call trees. If an instance of an exception for a particular method call was included in one of the captured call trees, the method name in the Exceptions table is preceded by a plus sign (+) or a minus sign (-). The plus sign indicates that when you click the row in the table, the entry expands to reveal additional rows with the exception class for each of the captured instances of the exception. The minus sign indicates that when you click the row in the table, the entry contracts so that the exception class row is hidden.
Table	If two or more instances of an exception class were included in the captured call trees, the exception class name in the Exceptions table is preceded by a plus sign (+) or a minus sign (-). The plus sign indicates that when you click the row in the table, the entry expands to reveal the latency for each of the captured instances for the selected exception class. The minus sign indicates that when you click the row in the table, the entry contracts so that the latency for the captured exception class is hidden. If only one instance of an exception class was included in the captured call trees, the exception class in the Exceptions table is not preceded a plus sign or minus sign. In this case, the table entry itself represents the single instance of the exception class and the value in the latency for the exception can be determined from the Call Trees tab. You can view the call tree for an exception listed in the Exceptions table by clicking on any row that contains an instance of an exceptions class. (A row that does not have a plus sign (+) or a minus (-) sign before the exception class or that only contains a latency value is an exception class instance.) When you click a row with an exception class instance, the profiler switches to the Call Tree tab and displays the call tree for the selected exception instance. The method call that generated the exception for the selected exception class is highlighted in blue in the call tree. For information on the Call Tree tab, see "Call Tree Tab Description" on page 225.

Collections Tab Description

The .NET Diagnostics Profiler can monitor your applications' memory usage using Lightweight Memory Diagnostics (LWMD). LWMD monitors the memory used by your applications by tracking the collections. The metrics from LWMD are displayed on the Collections tab. The memory metrics are shown in a graph of heap usage, and in tables that list the collections that are growing the fastest and that have become the largest. The Collections tab displays these problems, enabling identification of memory issues.

UI example	
To access	In the .NET Diagnostics Profiler, select the Collections tab.
Relevant tasks	"How to Access the .NET Diagnostics Profiler" on page 215

The following user interface elements are included:

UI Element	Description
Heap Usage Graph	The Heap Usage graph shows the memory that was committed and used at periodic sample intervals. (The default sample interval is 1 minute.) For each sample interval, a bar is displayed on the graph. • The height of the bar indicates the total amount of heap that was committed when the sample was taken. • The red portion of the bar indicates the amount of the heap that was committed and used when the sample was taken. • The green portion of the bar indicates the amount of the heap that was committed, but not used, when the sample was taken. Hold the mouse pointer over a sample's bar on the graph to display a tooltip showing the size of the heap that was used, followed by the size of the heap that was committed for the selected sample. By default, the LWMD process establishes a new baseline for measuring the growth of collections every hour. You can force a new baseline by clicking the **Force Baseline** link at the upper-right corner of the Heap Usage graph. When the .NET Diagnostics Profiler establishes a new baseline, a green line is inserted between the last sample of the previous baseline and the first sample of the next baseline to mark the point where the baseline was set. The calculation for the growth of collections that is used to determine which collections are included in the Collections by Growth table, is based on the number of collections added since the last baseline.
Samples and Collections Details Pane	Displays additional information about the sample selected in the Heap Usage graph, and about the collection selected from the collection tables. It contains the following information: **Sampled.** The date and time when the selected Heap Usage sample was taken. **Baselined.** The date and time of the last baseline prior to the sample being taken. **Contains.** The type of object contained in the selected collection. This information is displayed when you mouse over the Collections by Growth or Collections by Size tables. **Allocated In.** The method that allocated the selected collection. This information is displayed when you mouse over the Collections by Growth or Collections by Size tables.

UI Element	Description
Collections by Growth Table	The Collections by Growth table lists the top ten collections in relation to the growth in the number of objects contained in the collection since the last baseline. The top-ten list of collections changes from sample to sample as the growth rates for each collection fluctuate. When a new baseline is established, the growth rate is calculated in relation to the new baseline, so the list of collections can change significantly. The table contains the following information: **Growth.** The number of objects that were added to the collection since the last baseline. **Class.** The class name for the collection. To see details for the collection, hold the mouse pointer over the row in the table for the collection. The row is highlighted in pink and the details are displayed in the Samples and Collections Details pane.
Collections by Size Table	The Collections by Size table lists the top ten collections relative to the size of the collection for the selected Heap Usage sample. The size of a collection is based upon the total number of objects in the collection. The table contains the following information: **Size.** The total number of objects in the collection at the end of the sample period. **Class.** The class name for the collection. To see details for the collection, hold the mouse pointer over the row in the table for the collection. The row is highlighted in pink and the details are displayed in the Samples and Collections Details pane.

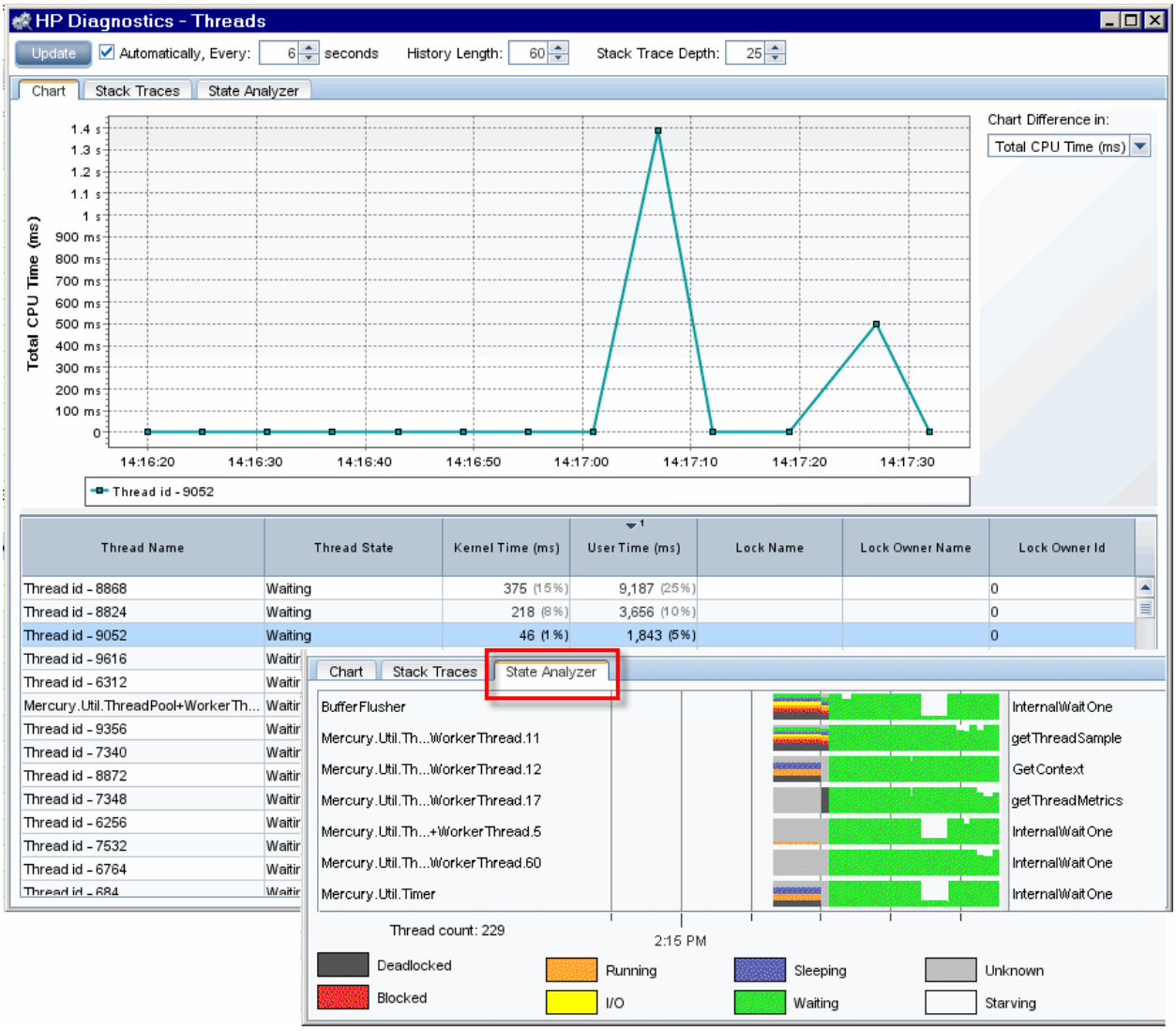
Threads Window Description

The Threads window displays thread performance metrics for the threads that are running in a .NET probed application and provides a way for you to capture stack traces for the running threads. There is also a thread state analyzer that displays approximate thread state distribution percentage for each thread.

This page can be useful for helping to diagnose the following situations:

- Incorrect thread pooling or attempting to do too much in a single thread.
- Performance problems caused by deadlocks or concurrency-related issues.
- Problems that go deep into the interactions with the OS kernel where you need to see the CPU time broken into user and kernel times.

The following is an example of the .NET Threads display.



To access	Select a .NET probe from the the .NET Probes or Probes view, then click **View Threads in New Window** from the Common Tasks area.

The following user interface elements are included:

UI Element	Description
Controls	Used to control how often the thread metrics are updated, maximum stack trace depth for each thread, and what kind of data is displayed for the thread processing in your application. When the Threads tab is updated, the information displayed on the tab is refreshed with the latest thread metrics. You control how often the Profiler updates the thread metrics on the Threads tab. **Update button.** Select the Update button and the Profiler refreshes the information in the graph and the thread table and captures stack traces. **Automatically, Every (Thread Metric Update Frequency).** Check this box to turn automatic updates on. Select the update interval from the spinner. The Profiler immediately begins refreshing the thread metrics displayed in this tab based on the update interval specified. Whenever the Profiler updates the Threads tab display, stack traces are captured for each of the threads listed in the thread table. You can control how many stack traces for each thread are displayed in the stack trace history. **History Length.** Select the number of samples to keep and display. **Stack Trace Depth.** Select the maximum stack trace depth collected for each sample for each thread.

UI Element	Description
Chart Tab	Charts the metric for the selected threads. You may chart the metrics for one or more of the threads listed in the threads table and you can select the metric that is to be charted for each thread. Select a thread in the thread table to have it's metric graphed in the chart. Diagnostics removes the metrics for any previously charted threads from the graph and charts the metric for the selected thread. The graph legend is updated to indicate the color with which the selected thread's metrics were charted. To chart additional threads in the graph along with any that you have already charted, select additional threads in the thread table. To select each additional thread one at a time, select each row in the thread table using **Ctrl-Click.** To select a range of threads, select the row in the thread table using **Shift-Click.** Diagnostics charts the metrics for the selected thread along with the metrics for all of the threads in the thread table that are between the selected threads and the newly selected thread. The graph legend is updated to indicate the colors with which the selected threads metrics were charted. To remove the metrics from the chart for selected threads, use **Ctrl-Click** to select the row in the thread table that contains the thread whose metrics you'd like to remove from the chart. **Chart difference in.** To select a metric to be charted for each thread, select the metric from the drop down menu. Diagnostics updates the graph to chart the indicated metric for each of the threads selected in the thread table.
Thread Table	The table shown below the chart lists the metrics for each thread. The following columns are displayed: **Thread Name.** The name of the captured thread. **Thread State.** The state of the thread at the last thread metric update interval. **Kernel Time (ms).** The portion of the CPU time during which the thread was executing in kernel mode. **User Time (ms).** The portion of the CPU time during which the thread was executing in user mode. The following data comes from the JVM: Lock Name, Lock Owner Name, Lock Owner Id. The table can also include columns for **Waited Time** and **Blocked Time** metrics if you enable them. To enable these metrics, set the **threads.contention.monitoring.enabled** property to `true` in the **<probe_install_dir>/etc/probe.properties** file. This setting may cause instability for some older JVMs.

UI Element	Description
Stack Traces Tab	Stack traces for the threads selected in the threads table are displayed when you have indicated that you want thread stack traces captured. The Stack Traces tab display is divided into two areas: **Captured Stack Traces.** List contains a list of the times when stack trace captures occurred. **Stack Trace Details.** Displays the stack traces that you indicated based on your selections from the stack trace capture list, the scope selection drop down, and the thread table. The **Stack Trace Details for** drop down allows you to control which thread's stack traces the Profiler displays in the Stack Trace details area. When you select **All Threads**, the stack traces for all threads are displayed in the stack trace details area. The selections made in the threads table do not impact the stack traces that are displayed in the stack trace details area when **All Threads** is selected. When you select **Selected Threads**, the stack traces displayed in the stack trace details area are limited to those for the threads that you select in the threads table in the Chart tab.

UI Element	Description
State Analyzer	The State Analyzer displays approximate thread state distribution percentage for each thread, over the specified time period. Each thread is represented by a single row. The left panel provides the thread name. The center panel provides the thread state data. The total height of the colored bar represents 100%. If a thread has been in more than one state during the observation period, multiple colors are used to display the corresponding states, proportional to the time spent in those states. For automatic updates, the observation period is the same as the configured refresh period. The right panel displays the current method name, with line number, if available. If the stack traces collected for the thread over the observation period are all the same, the method name is displayed using a bold font. If different stack traces were observed, the displayed method is the topmost common method for the collected stack traces, and its display uses a regular font. If no such common method could be found, nothing is displayed. The following thread states are presented by the Thread State Analyzer: **Deadlocked.** The thread participates in a deadlock cycle. **Blocked.** The thread is delayed (suspended) when trying to enter a Java monitor. This can happen when the thread tries to invoke a synchronized method, enter a synchronized block, or re-enter the Java monitor after being awaken from the waiting state, while another thread has not left the Java monitor yet. **Running.** The thread is actively consuming CPU time. **I/O.** The thread is performing an I/O operation. It does not use any CPU time. The notion of I/O covers not only the traditional operations on files or sockets, but also covers any multimedia or graphics operations. In general, the thread is waiting for an external (out-of-process) event. **Sleeping.** The thread is delayed after invoking the Thread.sleep() method. **Waiting.** The thread is delayed, usually having executed Object.wait(). However, threads can get into this state by other means. In general, the thread is waiting for an internal (in-process) event. **Starving.** The thread is runnable, it is not suspended by any I/O, wait(), or sleep() operation, but is not running. This can be caused by insufficient number of CPUs available, Garbage Collection pauses, excessive paging, or by a virtual machine guest OS experiencing a shortage of resources. **Unknown.** The Diagnostics Agent was unable to determine the state of the thread.

Send Documentation Feedback

If you have comments about this document, you can contact the documentation team by email. If an email client is configured on this system, click the link above and an email window opens with the following information in the subject line:

Feedback on .NET Agent Guide (Diagnostics 9.24)

Just add your feedback to the email and click send.

If no email client is available, copy the information above to a new message in a web mail client, and send your feedback to SW-doc@hp.com.

We appreciate your feedback!