Guest and HA Application Monitoring SDK Programming Guide

17 APR 2018
VMware vSphere 6.7
VMware ESXi 6.7

vmware®

You can find the most up-to-date technical documentation on the VMware website at:

https://docs.vmware.com/

If you have comments about this documentation, submit your feedback to

docfeedback@vmware.com

VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

Contents

About This Book

The *Guest and HA Application Monitoring SDK Programming Guide* provides information about developing applications using the VMware® Guest Application Programming Interface.

VMware provides several different software development kit (SDK) products. They target different developer communities and platforms. This guide is intended for developers who want to retrieve information about the virtual machine and host hardware where the application runs. The supported VMware platforms include ESXi 5.5, ESXi 6.0, ESXi 6.5, and ESXi 6.7.

Revision History

This book is revised with each release of the product or when necessary. A revised version can contain minor or major changes. Table 1 summarizes the significant changes in each version of this book.

Table 1. Revision History

Revision Date	Description
17 Apr 2018	For ESXi 6.7.
15 Nov 2016	For ESXi 6.5. Minor modifications to “Note on vm.cpu.contention.cpu” section.
16 May 2016	Corrected formulas for extended guest statistics; added `esxtop` comparison.
25 Mar 2016	Added section about `checksystem` utility to verify `glib` version.
22 Jun 2015	Documented the `vmware-appmonitor postAppState` option and arguments.
17 Apr 2015	Added new chapter on fetching extended guest statistics.
29 Sept 2014	Update for ESXi 6.0, with new section about security of Remote RPC.
19 Sept 2013	Update for ESXi 5.5, with new VMGuestAppMonitor_PostAppState function.
17 May 2012	Added vSphere HA Application Monitoring, changed version number for VMware Tools 9.0.
24 Aug 2011	Added information about compatibility with vSphere 5.0.
13 Jul 2010	No new information, but revised to note support for VMware ESX 4.1.
7 May 2009	Revised manual for VMware ESX version 4.0.
29 Nov 2007	No new information, but revised to note support for VMware ESX 3.5 and ESX 3i version 3.5.
18 Jul 2005	Initial release of the VMware Guest SDK providing support for VMware ESX 3.0.

Intended Audience

This book is intended for developers of software for vSphere high availability (HA) application monitoring, or for gathering statistics about guest operating systems.

VMware Technical Publications Glossary

VMware Technical Publications provides a glossary of terms that might be unfamiliar to you. For definitions of terms as they are used in VMware technical documentation go to http://www.vmware.com/support/pubs.

Document Feedback

VMware welcomes your suggestions for improving our documentation. Send your feedback to docfeedback@vmware.com.

Installing the Development Kit

1

Welcome to the VMware Guest and High Availability (HA) Application Monitoring software development kit.

This chapter includes the following topics:

About the SDK Contents

The Guest and HA Application Monitoring SDK is available as a tarball for Linux or a ZIP file for Windows. Both have a similar directory structure, shown in Table 1-1, with minor differences for compilation.

Table 1-1. Components of the SDK

Directory or Folder	Explanation of Contents
`bin/bin32` or `bin/win32` `bin/bin64` or `bin/win64`	Contains the `vmware-appmonitor` program, which controls the HA application monitoring heartbeat from the command line.
`docs/`	Contains the Guest SDK terms and conditions, and text of related Open Source licenses. Also contains sample code for HA application monitoring.

`docs/VMGuestAppMonitor/`
`samples/C` or
`samples/visualstudio`
`samples/java`
	The `samples/C` subdirectory (or `samples/visualstudio` subfolder) contains the `sample.c` (or `appmon.cpp`) program to demonstrate HA application monitoring API. Follow instructions in the README file to compile with `make` or with Visual Studio. The `samples/java` directory contains a Java native interface (JNI) implementation that builds on the C implementation. Again, see the README file.

`include/`	Header files for basic types, GuestAppMonitor and Guest libraries, and session ID.
`include/vmGuestLibTest.c`	Sample C program to run all the Guest library functions and return statistics. On Linux, use `gcc` to compile this program, and run it on an ESXi hosted virtual machine.
`lib/lib32` or `lib/win32` `lib/lib64` or `lib/win64`	Shared objects or DLL files and libraries for the Guest library, the Guest library for Java, and the HA application monitoring library.
`vmGuestLibJava` `vmGuestLibJava/doc`	JAR file and standard Javadoc for a prepackaged Java implementation of the Guest API. For a list of methods, browse `index.html` and see the VMGuestLibInterface page.

Displaying vSphere Guest Library Statistics

On a Linux virtual machine hosted by ESXi, go to the `include` directory and compile the `vmGuestLibTest.c` program. Run the output program `vmguestlibtest`.

```
gcc –g –o vmguestlibtest –ldl vmGuestLibTest.c
./vmguestlibtest
```

Guest statistics appear repeatedly until you interrupt the program.

Using the HA Application Monitoring SDK

This section provides a short introduction to the HA Application Monitoring SDK. You need information in Chapter 4 vSphere HA Application Monitoring to proceed further.

SDK function definitions and simple documentation are in the `vmGuestAppMonitorLib.h` include file.

Controlling the Application Monitoring Heartbeat

To run HA application monitoring programs, the virtual machine must be running on an ESXi host, and application monitoring must have been enabled when configuring HA.

You can enable heartbeats with the compiled `vmware–appmonitor` program. Usage is as follows:

```
vmware–appmonitor { enable | disable | markActive | isEnabled | getAppStatus | postAppState }
```

- `enable` – Enable application heartbeat so vSphere HA starts listening and monitoring the heartbeat count from this guest virtual machine. The heartbeats should be sent at least once every 30 seconds.
- `disable` – Disable the application heartbeat so vSphere HA stops listening to heartbeats from this guest.
- `markActive` – This starts sending the actual heartbeat every 30 seconds or less.
- `isEnabled` – Indicates whether the heartbeat monitoring was enabled.
- `getAppStatus` – Gets the status of the application, either Green, Red, or Gray.
- `postAppState` – Posts the state of the application. Arguments can be:
  - `appStateOk` – Sends an “Application State is OK” signal to the HA agent running on the host.
  - `appStateNeedReset` – Sends an “Immediate Reset” signal to the HA agent running on the host.

Library Path or Path Environment

On Linux, set your `LD_LIBRARY_PATH` environment to the install location of `GuestSDK/lib/lib32` or `lib64`. On Windows, you can set your `PATH` environment, but it is probably easier to copy `vmware–appmonitor` to the same folder as the DLL files.

Compiling the Sample Program on Linux

You need a C compiler and the `make` program.

Procedure

1. Go to the `docs/VMGuestAppMonitor/samples/C` directory.
2. Run the `make` command.

   On a 64-bit machine you might want to change `lib32` to `lib64` in the `makefile`.
3. Set `LD_LIBRARY_PATH` as described above.
4. Run the sample program. See below for program usage.

   ```
   ./sample
   ```

Compiling Sample Programs on Windows

You need Visual Studio 2008 or later.

Procedure

1. Go to the `docs/VMGuestAppMonitor/samples/visualstudio` folder.
2. Open the `appmon.vcproj` file and build the solution.
3. Click **Debug > Start Debugging** to run `appmon.exe`. See below for program usage.

Demonstrating the HA Application Monitoring API

The sample program enables HA application monitoring and sends a heartbeat every 15 seconds. After the program starts running, typing Ctrl+C displays three choices:

- s – stop sending heartbeats and exit the program. The virtual machine will reset.
- d – disable application monitoring and exit the program. This does not cause a reset.
- c – continue sending heartbeats.

Security of Remote RPC

Guest RPC is a communication channel between the guest operating system and its VMX, or virtual machine executable, the user space component of virtual infrastructure. The VMM, or virtual machine monitor, is the kernel space component.

In the ESXi 6.0 release, Guest RPC was reimplemented on top of VMCI Sockets.

To enforce security for both the Guest SDK and the HA Application Monitoring SDK, allowing only root and Administrator access to the functions provided by the SDK, on ESXi 6.0 hosts you can edit the `.vmx` file for the respective virtual machine and set the secure authentication parameter as follows:

```
guest_rpc.rpci.auth.app.APP_MONITOR = TRUE
```

If you do not need to enforce security and want to allow non-root and non-Administrator users to access functions in the Guest and HA Application Monitoring SDK, the secure authentication parameter must not appear in the `.vmx` file, or it must be set `FALSE`.

The Guest Programming API 2

The VMware Guest API provides functions that you can use in a program that runs in the guest operating system environment on a VMware ESXi host.

This chapter includes the following topics:

Overview of the vSphere Guest API

The vSphere Guest API provides functions that management agents and other software can use to collect data about the state and performance of a VMware virtual machine. The Guest API provides fast access to resource management information, without the need for authentication.

The Guest API provides read-only access. You can read data using the API, but you cannot send control commands. To send control commands, use the vSphere Web Services SDK. For more information, see the *VMware vSphere Web Services SDK Programming Guide* and the *VMware vSphere API Reference*, which are available on the VMware developer support Web site.

The version number of this Guest API release is 9.0 to match the version number of VMware Tools.

Supported Guest Operating Systems

The vSphere Guest API software runs on most Windows or Linux guest operating systems supported by the various versions of ESXi.

See the *VMware Compatibility Guide* for a list of supported guest operating system versions. This guide is now located at http://www.vmware.com/resources/compatibility in Web format.

The Guest API does not support the following guest operating system environments:

- Windows 95 and Windows 98.
- Windows NT 4.0. For Windows NT 4.0 you must use Guest SDK 3.5, which you can find by going to http://www.vmware.com/support/developer/guest-sdk and selecting an old release.

Required Library Versions

To help you verify the minimum library version, the `checksystem` utility is bundled in the Guest SDK package for Linux guest OS systems. After you extract the Guest SDK, the utility is available in the `GuestSDK/bin` subdirectory. If the guest OS has a compatible `glibc`, the `checksystem` utility prints this message:

```
This version of GuestSDK is compatible with the version of glibc in this system.
```

If the Linux guest OS does not have a compatible `glibc`, this error message appears:

```
This version of GuestSDK requires version 2.5 or later of glibc. For this system,
use version 9.10.x of GuestSDK from https://www.vmware.com/support/developer/guest-sdk/
```

Virtual Machine Statistics

With the Guest API, you can monitor various statistics about the virtual machine. You can use this information to retrieve scheduling and resource usage information about the environment. With the help of these statistics, a virtual machine can immediately react to changes in its virtual environment at the application layer.

The information you can retrieve by using the vSphere Guest API includes:

- Amount of memory reserved for the virtual machine.
- Amount of memory being used by the virtual machine.
- Upper limit of memory available to the virtual machine.
- Number of memory shares assigned to the virtual machine.
- Maximum speed of the virtual machine's CPU.
- Reserved rate at which the virtual machine is allowed to execute. An idling virtual machine might consume CPU cycles at a much lower rate.
- Number of CPU shares assigned to the virtual machine.
- Elapsed time since the virtual machine was last powered on or reset.
- CPU time consumed by a particular virtual machine. When combined with other measurements, you can estimate how fast the virtual machine's CPUs are running compared to the host CPUs.

Important The API uses a handle that provides access to the statistics. The handle also is a mechanism to determine whether the API can provide accurate information. (Certain events, such as migrating a virtual machine with VMotion™, temporarily make it impossible to provide accurate information.)

How to Use the vSphere Guest API

The vSphere Guest API defines functions and data types that you use to extract virtual machine data. This section covers the following topics:

- vSphere Guest API Runtime Components
- vSphere Guest API Data Types
- vSphere Guest API Functions
- vSphere Guest API Error Codes

vSphere Guest API Runtime Components

To use the vSphere Guest API, the runtime components must be installed in the guest operating system. The runtime components are dynamically loaded binary modules for 32-bit and 64-bit guests. When you install VMware Tools, the vSphere Guest API runtime components are installed as well. You can also download them from http://www.vmware.com/download/sdk/guest_sdk.html.

To make the vSphere Guest API functions available to your program, use your program's standard methods to load the library.

- In a Windows guest operating system, the library file is `vmGuestLib.dll`. The import library file is `vmGuestLib.lib`.
- In a Linux guest operating system, the library file is `libvmGuestLib.so`.

If you are using a Security-Enhanced Linux (SELinux) guest OS, its security policies might interfere with dynamic loading of `libvmGuestLib.so`. Refer to documentation about your SELinux policy configuration

---

The vSphere Guest SDK includes the test program `vmGuestlibTest.c`. If you are using a Windows environment, you must rebuild the test program. The `vmGuestLib.dll` library file is a non-Unicode DLL. In Microsoft Visual Studio, build the test program `vmGuestlibTest.c` as a non-Unicode executable file so that the program can access the DLL at runtime.

Enabling and Disabling the Runtime Components

The vSphere Guest API runtime components are enabled by default (`disable = "FALSE"`). To disable the runtime components, use the configuration editor in the vSphere Client to edit the configuration file for the virtual machine. The virtual machine must be powered off before you can use the configuration editor.

1. In the vSphere Client window, right-click the virtual machine in the machine list.
2. In the drop-down menu, select Edit Settings.
3. In the Virtual Machine Properties window, click the Options tab.
4. In the list of "Advanced" settings, select General.
5. Click Configuration Parameters.

6 In the Configuration Parameters window, add the following line or, if the file already contains the disable configuration setting, set the value to **TRUE**:

```
isolation.tools.guestlibGetInfo.disable = "TRUE"
```

The default value for the disable setting is **FALSE**. The default setting enables the runtime components. Reinstalling VMware Tools does not affect the disable setting. If you disable the vSphere Guest API and reinstall VMware Tools, the vSphere Guest API remains unavailable until you change the configuration setting `guestLibGetInfo.disable` to **FALSE**.

vSphere Guest API Data Types

The vSphere Guest API uses the data types listed in Table 2-1 to support access to virtual machine data.

Data Type	Description
`VMGuestLibHandle`	Reference to virtual machine data. `VMGuestLibHandle` is defined in `vmGuestLib.h`.
`VMSessionID`	Unique identifier for a session. The session ID changes after a virtual machine is migrated using VMotion, suspended and resumed, or reverted to a snapshot. Any of these events is likely to render any information retrieved with this API invalid. You can use the session ID to detect those events and react accordingly. For example, you can refresh and reset any state that relies on the validity of previously retrieved information. Use `VMGuestLib_GetSessionId` to obtain a valid session ID. A session ID is opaque. You cannot compare a virtual machine session ID with the session IDs from any other virtual machines.You must always call `VMGuestLib_GetSessionId` after calling `VMGuestLib_UpdateInfo`. `VMSessionID` is defined in `vmSessionId.h`.
`VMGuestLibError`	Status code that indicates success or failure. Each function returns a `VMGuestLibError` code. For information about specific error codes, see vSphere Guest API Error Codes. `VMGuestLibError` is an enumerated type defined in `vmGuestLib.h`.

vSphere Guest API Functions

The vSphere Guest SDK contains the header file `vmGuestLib.h`. This file declares the functions and data types that you use to call the vSphere Guest API. The following sections describe the vSphere Guest API functions:

- Calling Context Functions
- Accessor Functions (Virtual Machine)

Calling Context Functions

The vSphere Guest API provides a set of functions that initialize and manipulate the context where the Guest API operates. Before your application can use the accessor functions to retrieve information about a virtual machine, use the following functions to initialize the vSphere Guest API environment.

1 Call the `VMGuestLib_OpenHandle` function to obtain a handle for accessing information about the virtual machine. The guest library handle is a parameter to every Guest API function.

2 Call the `VMGuestLib_UpdateInfo` function to update the information available through the handle.

3 Call the `VMGuestLib_GetSessionId` function to retrieve a session ID.

About Context Functions

#GUID-3276EBD2-DFF5-4F1C-A5C8-091DA91A0645/ID-3875-00000233 shows a C code fragment that illustrates the function calls for initialization. (The code fragments in this section do not perform error handling. For information about error handling, see vSphere Guest API Error Codes.)

Initializing the vSphere Guest API Environment

```
VMGuestLibHandle glHandle;
VMGuestLibError glError;
VMSessionId sid = 0;
glError = VMGuestLib_OpenHandle(&glHandle);
glError = VMGuestLib_UpdateInfo(glHandle);
glError = VMGuestLib_GetSessionId(glHandle, &sid);
```

You can use the session ID to detect changes that invalidate previously retrieved data. #unique_21/unique_21_Connect_42_ID-3875-0000023F shows a code fragment that illustrates how to use the session ID to detect stale data. (The `ResetStats` function in the following fragment represents application code to handle the session change.)

Detecting Stale Data

```
VMGuestLibHandle glHandle;
VMGuestLibError glError;
VMSessionId oldSid;
VMSessionId newSid;

/* [...code here would access data based on an existing, valid session ID (oldSid)...] */

/* Update the library, get the session ID, and compare it to the previous session ID */
glError = VMGuestLib_UpdateInfo(glHandle);
glError = GuestLib_GetSessionId(glHandle, &newSid);
if (oldSid != newSid) {
    ResetStats();
    oldSid = newSid;
}
```

Table 2-2 lists the context functions for creating and releasing handles, updating information, and obtaining session IDs.

Function	Description
`VMGuestLib_OpenHandle`	Gets a handle for use with other vSphere Guest API functions. The guest library handle provides a context for accessing information about the virtual machine. Virtual machine statistics and state data are associated with a particular guest library handle, so using one handle does not affect the data associated with another handle.
`VMGuestLib_CloseHandle`	Releases a handle acquired with `VMGuestLib_OpenHandle`.

VMGuestLib_UpdateInfo	Updates information about the virtual machine. This information is associated with the VMGuestLibHandle. VMGuestLib_UpdateInfo requires similar CPU resources to a system call and therefore can affect performance. If you are concerned about performance, minimize the number of calls to VMGuestLib_UpdateInfo. If your program uses multiple threads, each thread must use a different handle. Otherwise, you must implement a locking scheme around update calls. The vSphere Guest API does not implement internal locking around access with a handle.
VMGuestLib_GetSessionId	Retrieves the VMSessionID for the current session. Call this function after calling VMGuestLib_UpdateInfo. If VMGuestLib_UpdateInfo has never been called, VMGuestLib_GetSessionId returns VMGUESTLIB_ERROR_NO_INFO.

Accessor Functions (Virtual Machine)

Accessor functions retrieve information about a virtual machine. When you call an accessor function, you pass a guest library handle (type VMGuestLibHandle) to the function. If the function is successful, it returns the requested data as an output parameter. The function return value is an error code (type VMGuestLibError) that indicates success or failure.
#unique_21/unique_21_Connect_42_ID-3875-0000028A shows a C code fragment that illustrates an example of calling VMGuestLib_GetCpuLimitMHz to retrieve the processor limit available to the virtual machine.

Using an Accessor Function

```
uint32 cpuLimitMHz = 0;
glError = VMGuestLib_GetCpuLimitMHz(glHandle, &cpuLimitMHz);
```

When a call completes successfully, the function returns the value VMGUESTLIB_ERROR_SUCCESS. If the call is unsuccessful, the error code name contains an appropriate description. For details, see vSphere Guest API Error Codes.

Call VMGuestLib_UpdateInfo once to refresh all statistics before calling an accessor function or a series of accessor functions.

Function	Description
VMGuestLib_GetCpuLimitMHz	Retrieves the upper limit of processor use in MHz available to the virtual machine. For information about setting the CPU limit, see Limits and Reservations.
VMGuestLib_GetCpuReservationMHz	Retrieves the minimum processing power in MHz reserved for the virtual machine. For information about setting a CPU reservation, see Limits and Reservations.
VMGuestLib_GetCpuShares	Retrieves the number of CPU shares allocated to the virtual machine. For information about how an ESXi host uses CPU shares to manage virtual machine priority, see the *vSphere Resource Management Guide*.
VMGuestLib_GetCpuStolenMs	Retrieves the number of milliseconds that the virtual machine was in a ready state (able to transition to a run state), but was not scheduled to run.

VMGuestLib_GetCpuUsedMs	Retrieves the number of milliseconds during which the virtual machine has used the CPU. This value includes the time used by the guest operating system and the time used by virtualization code for tasks for this virtual machine. You can combine this value with the elapsed time (VMGuestLib_GetElapsedMs) to estimate the effective virtual machine CPU speed. This value is a subset of elapsedMs.
VMGuestLib_GetElapsedMs	Retrieves the number of milliseconds that have passed in the virtual machine since it last started running on the server. The count of elapsed time restarts each time the virtual machine is powered on, resumed, or migrated using VMotion. This value counts milliseconds, regardless of whether the virtual machine is using processing power during that time. You can combine this value with the CPU time used by the virtual machine (VMGuestLib_GetCpuUsedMs) to estimate the effective virtual machine CPU speed. cpuUsedMS is a subset of this value.
VMGuestLib_GetHostProcessorSpeed	Retrieves the speed of the ESXi host's physical CPU in MHz.
VMGuestLib_GetMemActiveMB	Retrieves the amount of memory the virtual machine is actively using—its estimated working set size.
VMGuestLib_GetMemBalloonedMB	Retrieves the amount of memory that has been reclaimed from this virtual machine by the vSphere memory balloon driver, which is also called the "vmmemctl" driver.
VMGuestLib_GetMemLimitMB	Retrieves the upper limit of memory that is available to the virtual machine. For information about setting a memory limit, see Limits and Reservations.
VMGuestLib_GetMemMappedMB	Retrieves the amount of memory that is allocated to the virtual machine. Memory that is ballooned, swapped, or has never been accessed is excluded.
VMGuestLib_GetMemOverheadMB	Retrieves the amount of "overhead" memory associated with this virtual machine that is currently consumed on the host system. Overhead memory is additional memory that is reserved for data structures required by the virtualization layer.
VMGuestLib_GetMemReservationMB	Retrieves the minimum amount of memory that is reserved for the virtual machine. For information about setting a memory reservation, see Limits and Reservations.
VMGuestLib_GetMemSharedMB	Retrieves the amount of physical memory associated with this virtual machine that is copy-on-write (COW) shared on the host.
VMGuestLib_GetMemSharedSavedMB	Retrieves the estimated amount of physical memory on the host saved from copy-on-write (COW) shared guest physical memory.
VMGuestLib_GetMemShares	Retrieves the number of memory shares allocated to the virtual machine. For information about how an ESXi host uses memory shares to manage virtual machine priority, see the *vSphere Resource Management Guide*.
VMGuestLib_GetMemSwappedMB	Retrieves the amount of memory that has been reclaimed from this virtual machine by transparently swapping guest memory to disk.
VMGuestLib_GetMemTargetSizeMB	Retrieves the size of the target memory allocation for this virtual machine.

`VMGuestLib_GetMemUsedMB`	Retrieves the estimated amount of physical host memory currently consumed for this virtual machine's physical memory.
`VMGuestLib_GetResourcePoolPath`	Retrieves the path name of the resource pool affiliated with the virtual machine on the ESXi host where it is running. VMGuestLib_GetResourcePoolPath uses an additional parameter to determine the size of the path name output string buffer. `VMGuestLibError VmGuestLib_GetResourcePoolPath( VMGuestLibHandle handle, size_t *bufferSize, char *pathBuffer );` `handle` is an input parameter specifying the guest library handle. `bufferSize` is an input/output parameter. It is a pointer to the size of the pathBuffer in bytes. If bufferSize is not large enough to accommodate the path (including a NULL terminator), the function returns VMGUESTLIB_ERROR_BUFFER_TOO_SMALL. In this case, the function uses the bufferSize parameter to return the number of bytes required for the string. `pathBuffer` is an output parameter. It is a pointer to a buffer that receives the resource pool path string. The path string is NULL-terminated. For information about using resource pools, see the *vSphere Resource Management Guide*.

For more information about ESXi resource management, see the *vSphere Resource Management Guide*, available on the VMware Web site.

Limits and Reservations

You use the Guest API to retrieve information about limits and reservations for CPU and memory. To set limits and reservations, you can use the vSphere (Web) Client or the vSphere API. Setting limits and reservations on a virtual machine ensures that resources are available to that machine and to other virtual machines that draw resources from the same resource pool.

To use the vSphere Client to set limits or reservations:

1 In the vSphere Client window, click the Resource Allocation tab.

2 In either the CPU or Memory section, click Edit.

3 In the Virtual Machine Properties Window, click the Resources tab.

4 Select either the CPU or Memory setting.

5 Use the slider controls to set limits or reservations.

For more information, see online help for the vSphere Client.

To use the vSphere API to set limits or reservations, call the ReconfigVM_Task method. In the method call, use the VirtualMachineConfigSpec data object to set the cpuAllocation or memoryAllocation property. These properties are of type ResourceAllocationInfo, which has limit and reservation properties. For more information, see the VMware vSphere API Reference Documentation.

vSphere Guest API Error Codes

Each vSphere Guest API function returns an error code. If successful, the returned error code is `VMGUESTLIB_ERROR_SUCCESS`. If the function is unable to complete its task, the error code provides information for diagnosing the problem.

Use the `VMGuestLib_GetErrorText` function to retrieve the error text associated with a particular error code. When you call the function, pass the error code to the function; `VMGuestLib_GetErrorText` returns a pointer to a string containing the error text.

#unique_22/unique_22_Connect_42_ID-3875-0000035B shows error handling. The C code fragment declares a guest library handle and calls the function `VMGuestLib_OpenHandle`. If the call is not successful, the code calls `VMGuestLib_GetErrorText` and displays the error text.

Error Handling

```
VMGuestLibHandle glHandle;
glError = VMGuestLib_OpenHandle(&glHandle);
if (glError != VMGUESTLIB_ERROR_SUCCESS) {
   printf("OpenHandle failed: %s\n", VMGuestLib_GetErrorText(glError));
}
```

Table 2-4 lists all error codes defined for the vSphere Guest API.

Error Code	Description
`VMGUESTLIB_ERROR_SUCCESS`	The function has completed successfully.
`VMGUESTLIB_ERROR_OTHER`	An error has occurred. No additional information about the type of error is available.
`VMGUESTLIB_ERROR_NOT_RUNNING_IN_VM`	The program making this call is not running on a VMware virtual machine.
`VMGUESTLIB_ERROR_NOT_ENABLED`	The vSphere Guest API is not enabled on this host, so these functions cannot be used. For information about how to enable the library, see Calling Context Functions.
`VMGUESTLIB_ERROR_NOT_AVAILABLE`	The information requested is not available on this host.
`VMGUESTLIB_ERROR_NO_INFO`	The handle data structure does not contain any information. You must call `VMGuestLib_UpdateInfo` to update the data structure.
`VMGUESTLIB_ERROR_MEMORY`	There is not enough memory available to complete the call.
`VMGUESTLIB_ERROR_BUFFER_TOO_SMALL`	The buffer is too small to accommodate the function call. For example, when you call `VMGuestLib_GetResourcePoolPath`, if the path buffer is too small for the resulting resource pool path, the function returns this error. To resolve this error, allocate a larger buffer.
`VMGUESTLIB_ERROR_INVALID_HANDLE`	The handle that you used is invalid. Make sure that you have the correct handle and that it is open. It might be necessary to create a new handle using `VMGuestLib_OpenHandle`.
`VMGUESTLIB_ERROR_INVALID_ARG`	One or more of the arguments passed to the function were invalid.
`VMGUESTLIB_ERROR_UNSUPPORTED_VERSION`	The host does not support the requested statistic.

Tools for Extended Guest Statistics

3

As of the vSphere 6.0 release, VMware Tools have expanded support for guest operating system statistics. Rather than offering a fixed set of statistics after calling `VMGuestLib_UpdateInfo()`, a larger and extensible set of statistics are available by calling `VMGuestLib_StatGet()`, or by using the CLI.

This chapter includes the following topics:

Introduction to Statistics Fetch

The fetch-statistics facility returns semi-structured data in a variety of formats for use within a the guest OS. Four encodings are supported:

- text – simple key-value pairs
- XML – eXtensible Markup Language fragment
- JSON – JavaScript Object Notation
- YAML – Yet Another Markup Language, a human-readable JSON superset intended for data transmission

Statistics are accurate within the current session only. A session represents a powered-on virtual machine running on a single ESXi host. Sessions may be discontinuous across session changes, so monotonically increasing metrics can suddenly decrease. New sessions result from suspending a virtual machine, reverting to a snapshot, vMotion, or certain types of hot-plug operations. It is up to the consumer of statistics to notice that the session has changed and deal with numerical discontinuities.

Guest OS statistics are supported for troubleshooting and support only. Specifically:

- Individual metrics are not guaranteed to be forward or backward compatible. Programs using these metrics are expected to tolerate incompatibility, and are cautioned against programming practices that could break when a metric behaves differently.

- Metrics might be added, removed, or modified to have a different meaning. For example: `co-stop` time changed meaning between ESXi 4.1 and ESXi 5.0.
- VMware reserves the right to make changes, but will try to avoid gratuitous changes. Most changes will still provide equivalent information, but possibly in a different form that better reflects the ESXi host implementation providing the metrics.

This support declaration is similar to the compatibility guarantee for the `esxtop` command, whose statistics the Guest SDK reflects. See https://communities.vmware.com/docs/DOC-9279 for details about `esxtop`.

Prerequisites

Before you start, install the latest version of VMware Tools in all relevant virtual machines.

For extended guest statistics, you must have VMware Tools version 9.10 or later installed in a virtual machine running on an ESXi 6.0 or later host. There is no minimum virtual hardware version. Version 9.10 was released with vSphere 6.0. Earlier ESXi hosts return no data for extended statistics requests, and earlier VMware Tools lack extended statistics. If statistics are unavailable, a code indicates that the requested item is unsupported.

Guest Statistics Interfaces

You have a choice of three interfaces to fetch the new statistics: the guest SDK library, CLI, or raw RPC.

Guest SDK Library

The Guest SDK library now offers two new functions, get and free.

VMGuestLib_StatGet

```
/* Semi-structured hypervisor statistics collection, for troubleshooting.
 */
VMGuestLibError
VMGuestLib_StatGet(const char *encoding, // IN
const char *stat,    // IN
char **reply,        // OUT
size_t *replySize); // OUT
```

- `encoding` – ‘‘text’’ or “xml” or “json” or “yaml” – if not specified, “text” is the default.
- `stat` – the statistic to print. See examples below.
- `reply` – a pointer to be set with a buffer containing the formatted reply. All current formats return null-terminated C strings, but future formats may not; the caller should treat the buffer as binary unless the format is known. The buffer must later be freed by a call to `VMGuestLib_StatFree()`.
- `replySize` – a pointer to receive the size of data in the buffer.

VMGuestLib_StatFree

To free the memory returned by `VMGuestLib_StatGet`, call `VMGuestLib_StatFree()`.

```
void
VMGuestLib_StatFree(char *reply, size_t replySize);
```

- `reply` – the pointer that was supplied by the `reply` parameter of `VMGuestLib_StatGet()`.
- `replySize` – the size that was supplied by the `replySize` parameter of `VMGuestLib_StatGet()`.

Example: C code with StatGet and StatFree functions shows these two function calls used in a sample routine:

Example: C code with StatGet and StatFree functions

```
/*
 * Retrieves semi-structured statistics on ESXi host.
 */
static int
StatGetRaw(const char *encoding, // IN
const char *stat,   // IN
const char *param)  // IN
{
   int exitStatus = EXIT_SUCCESS;
   VMGuestLibError glError;
   char *result = NULL;
   size_t resultSize = 0;
   char *arg = g_strdup_printf("%s %s", stat, param);

   glError = VMGuestLib_StatGet(encoding, arg, &result, &resultSize);
   if (glError != VMGUESTLIB_ERROR_SUCCESS) {
      exitStatus = EX_TEMPFAIL;
   } else {
      g_print("%*s", (int)resultSize, result);
   }
   VMGuestLib_StatFree(result, resultSize);
   g_free(arg);
   return exitStatus;
}
```

Command Line Interface

The command name varies with each of the three OS types available, but all three have a –v option to show the VMware Tools version number.

```
Linux:   vmware-toolbox-cmd stat raw <encoding> <stat>
Mac:     vmware-tools-cli stat raw <encoding> <stat>
Windows: VMwareToolboxCmd.exe stat raw <encoding> <stat>
```

These commands are provided for debugging and scripting; the implementation is a wrapper on top of the `VMGuestLib_StatGet()` call described above.

Raw RPC interface

The raw RPC interface varies for one of the three OS types available.

```
Linux:       vmtoolsd --cmd="guestlib.stat.get <encoding> <stat>"
Mac:         vmtoolsd --cmd="guestlib.stat.get <encoding> <stat>"
Windows:     vmtoolsd.exe --cmd="guestlib.stat.get <encoding> <stat>"
```

This is a raw form of the statistics API. Function calls or CLI are preferred.

Fetching a List of Statistics

To fetch a list of statistics, make a query with no statistics name supplied (in other words, no argument, `NULL`, or `""`). The returned buffer contains a list of available metric categories. The list can be emitted in any of four supported formats. The `text` format is recommended for user interaction; machine-parsable formats `xml`, `json`, and `yaml` are recommended for program interaction.

The following examples use the CLI to demonstrate. The C language API works similarly. In the first example, the virtual machine has three disks and two network interfaces.

Example: Return list of statistics

```
$ vmware-toolbox-cmd stat raw
session
host
resources
vscsi ide0:0
vscsi scsi0:0
vscsi scsi0:2
vnet 00:0c:29:1e:23:f3
vnet 00:0c:29:1e:23:f4
```

The second example fetches a specific statistic (vscsi scsi0:0) taken from the first example.

Example: Get I/O statistics for a device

```
$ vmware-toolbox-cmd stat raw text vscsi scsi0:0
num.reads = 12605
num.writes = 1039
size.reads = 533612032
size.writes = 14279680
latency.reads = 1944173239
latency.writes = 102025122
```

Metadata Fields

When appearing in machine-parsable formats, numeric statistics have these self-describing metadata fields:

`units` – What the numbers mean. Example values:

`packets`, `bytes`, `KB` for kilobytes, `us` for microseconds, and others as appropriate.

`type` – Indicates how to interpret the number.

`static` – The value is for configuration and is not expected to change frequently.
For instance: a reservation, or the maximum CPU speed.

`instant` – Instantaneous measurement of the current value. Expected to go up or down over time. For instance: current memory usage.

`accumulate` – Continuous sum of all data over time. Non-decreasing. The statistics consumer must compute the difference between two values at different times and convert it into a rate. For instance: CPU time used, bytes sent, memory swapped in.

Four examples below show the same statistic, encoded in different formats:

Example: Text

```
$ vmware-toolbox-cmd stat raw text vnet 00:0c:29:1e:23:f3
size.tx = 38137
num.rx = 10920
size.rx = 1312789
reservation = 0
limit = -1
```

Example: YAML

```
$ vmware-toolbox-cmd stat raw yaml vnet 00:0c:29:1e:23:f3
num.tx:
    type: accumulate
    units: packets
    value: 209
```

```
size.tx:
    type: accumulate
    units: bytes
    value: 38137
num.rx:
    type: accumulate
    units: packets
    value: 10920
size.rx:
    type: accumulate
    units: bytes
    value: 1312789
reservation:
    type: static
    units: MBps
    value: 0
limit:
    type: static
    units: MBps
    value: -1
```

Example: XML (formatted here for presentation)

```
$ vmware-toolbox-cmd stat raw xml vnet 00:0c:29:1e:23:f3
<metrics session="4004861987670969122">
<metric name="num.tx" type="accumulate" units="packets">209</metric>
<metric name="size.tx" type="accumulate" units="bytes">38137</metric>
<metric name="num.rx" type="accumulate" units="packets">10992</metric>
<metric name="size.rx" type="accumulate" units="bytes">1322161</metric>
<metric name="reservation" type="static" units="MBps">0</metric>
<metric name="limit" type="static" units="MBps">-1</metric>
</metrics>
```

Example: JSON (formatted here for presentation)

```
$ vmware-toolbox-cmd stat raw json vnet 00:0c:29:1e:23:f3
{"num.tx":{"type":"accumulate","units":"packets","value":209},
 "size.tx":{"type":"accumulate","units":"bytes","value":38137},
 "num.rx":{"type":"accumulate","units":"packets","value":11068},
 "size.rx":{"type":"accumulate","units":"bytes","value":1331791},
 "reservation":{"type":"static","units":"MBps","value":0},
 "limit":{"type":"static","units":"MBps","value":-1}}
```

Metrics Examples

The examples below use the default “text” format for readability. For actual programming, you will probably use one of the machine-parsable formats (XML, JSON, or YAML).

Fetch Available Statistics

The statistics command with no parameters fetches a list of available statistics:

```
$ vmware-toolbox-cmd stat raw
session
host
resources
vscsi scsi0:0
vnet 00:0c:29:1e:23:f3
```

`session` – the current session. This changes infrequently. Session changes can cause discontinuities.

`host` – information about the current hypervisor and hardware the virtual machine is running on.

`resources` – the CPU and memory usage of this virtual machine.

`vscsi` <name> – storage statistics for a specific virtual disk, where <name> is the internal name in the virtual machine's configuration file. The library does not aggregate storage statistics across disks.

`vnet` <Ethernet Address> – network statistics for a specific virtual NIC, where <Ethernet Address> is the MAC address currently programmed into the virtual NIC. The library does not aggregate network statistics across NICs.

For information about a specific statistic, enter its name.

Get Session Information

The statistics command with parameters fetches specific statistics:

```
$ vmware-toolbox-cmd stat raw text session
session = 4004861987670969122
uptime = 1036293956
version = VMware ESXi 6.0.0 build-12345
provider =
uuid.bios = 56 4d 2c 53 43 56 66 8e-7c 05 fd 7e 51 1e 23 f3
```

Sessions change for a virtual machine with power on, suspend, revert to snapshot, vMotion, reset, or when it experiences some type of hot-plug. Statistics might be discontinuous across a session change. The reason for a session change, or whether session change constitutes a host change, is not exposed to virtual machines.

`session` – a cryptographically strong random number indicating the current session. Expected to contain at least 63 bits of entropy. Changes with every session.

`uptime` – microseconds since the last session change, as measured by the host.

`version` – string representation of the hypervisor version. Not expected to be parsed. A guest OS should never change its behavior based on the hypervisor version or build number.

`provider` – string representing the provider. Set by the ExtraConfig `tools.guestlib.stat.provider` and intended for use by vSphere providers such as vCloud Air. Opaque contents defined by the provider.

`uuid.bios` – the virtual machine's SMBIOS UUID, cached at boot time. VMware maintains the SMBIOS UUID as a component of guest licensing; migrating a VM retains the same UUID while cloning a VM generates a different UUID. The algorithm to generate a new UUID varies from release to release. The vSphere API permits changing a VM's UUID. Although vSphere prevents unintentional duplication of UUID, it allows deliberate UUID duplication, because legitimate workflows (such as lab environments) often require duplicated UUIDs.

Host Hardware

In the next example, only the first two values are provided by default. A virtual machine with ExtraConfig `tools.guestlib.enableHostInfo = TRUE` (a non-default setting) supplies the remaining values.

```
$ vmware-toolbox-cmd stat raw text host
host.cpu.processorMHz = 2399
host.cpu.coresPerPkg = 4
host.cpu.packages = 2
host.cpu.cores = 8
host.cpu.threads = 16
host.dmi.product = ProLiant ML350 G6
host.dmi.vendor = HP
```

`host.cpu.processorMHz` – nominal processor speed. Other metrics, such as `vm.cpu.used` below, are normalized to this speed.

`host.cpu.coresPerPkg` – actual cores per socket, not including hyperthreads. Useful for determining cache effects and other aspects of socket sharing. Information is also available with CPUID instruction. Note that vSphere does not implement virtual hyperthreads.

`host.cpu.packages` – number of CPU sockets on the host (non-default).

`host.cpu.cores` – number of cores on the host across all sockets, not including hyperthreads (non-default).

`host.cpu.threads` – number of logical CPUs on the host across all sockets, including hyperthreads (non-default).

`host.dmi.product` – "product" field in the host SMBIOS data (non-default).

`host.dmi.vendor` – "vendor" field in the host SMBIOS data (non-default).

Host information (`dmi.product` and `dmi.vendor`) and total capacity are hidden by default, because this information is considered sensitive and not relevant to virtual machine execution. VMware discourages use of such information, but permits it to be made available to help with support.

CPU and Memory Statistics

For implementation reasons, a virtual machine tracks CPU and memory resources slightly differently. CPU resources, including NUMA, indicate virtualization overhead, shown with `vm.` prefix. Memory resources are broken out by guest memory, shown with `guest.` prefix, and by overhead memory, with `ovhd.` prefix. Future implementations may add additional metrics.

This example shows various CPU and memory statistics:

```
$ vmware-toolbox-cmd stat raw text resources
vm.cpu.reserved = 4798
vm.cpu.limit = 11995
vm.cpu.used = 224057517
vm.cpu.contention.cpu = 65606184
vm.cpu.contention.mem = 1488848
vm.numa.local = 1837248
vm.numa.remote = 0
guest.mem.reserved = 204800
guest.mem.limit = 1536000
guest.mem.mapped = 1810144
guest.mem.consumed = 1521680
guest.mem.swapped = 3236
guest.mem.ballooned = 27104
guest.mem.swapIn = 3416
guest.mem.swapOut = 6588
ovhd.mem.swapped = 0
ovhd.mem.swapIn = 0
ovhd.mem.swapOut = 0
```

`vm.cpu.reserved` – (static) MHz of current CPU type reserved. Covers all virtual CPU plus overheads, so for example a 2 virtual CPU machine would need 2x `host.cpu.processorMHz` to be fully reserved. Overheads are insignificant except during transient conditions such as taking a backup snapshot or during a vMotion. Default 0.

`vm.cpu.limit` – (static) MHz that the virtual machine will not exceed. Default –1 means unlimited.

`vm.cpu.used` – (cumulative) microseconds of CPU time used by this virtual machine. Equivalent to `esxtop %USED`. See Comparison to esxtop for details.

`vm.cpu.contention.cpu` = (cumulative) CPU time the virtual machine could have run, but did not run due to CPU contention. This metric includes time losses due to hypervisor factors, such as overcommit. Specific sources of contention vary widely from release to release. See Comparison to esxtop for details about calculating CPU contention.

`vm.cpu.contention.mem` – (cumulative) CPU time the virtual machine could have run, but did not run due to memory contention. This metric includes losses due to swapping. Equivalent to `esxtop %SWPWT`.

`vm.numa.local` – (instantaneous) KB of memory currently local, sum across the VM's NUMA nodes.

`vm.numa.remote` – (instantaneous) KB of memory currently remote, sum across the VM's NUMA nodes.

`guest.mem.reserved` – (static) KB of memory reserved for the guest OS. This indicates memory that will never be ballooned or swapped. Default is 0.

`guest.mem.limit` – (static) KB of memory the guest must operate within. Default –1 means unlimited.

`guest.mem.mapped` – (instantaneous) KB of memory currently mapped into the guest; that is, memory the guest can access with zero read latency. This metric represents memory use from a guest perspective.

guest.mem.consumed – (instantaneous) KB of memory used to provide current mapped memory. This might be lower than mapped due to ballooning, memory sharing, or future optimizations. This metric represents memory use from a host perspective. The difference between guest.mem.mapped and guest.mem.consumed is additional memory made available due to hypervisor optimizations.

guest.mem.swapped – (instantaneous) KB of memory swapped to disk. A fully reserved virtual machine should never see memory swapped out in steady-state usage. Transient conditions, such as resume from memory-included snapshot, might show some swap usage.

guest.mem.ballooned – (instantaneous) KB of memory deliberately copied on write (COWed) to zero in the guest OS, to reduce memory usage.

guest.mem.swapIn – (cumulative) KB of memory swapped in for the current session.

guest.mem.swapOut – (cumulative) KB of memory swapped out for the current session.

ovhd.mem.swapped – (instantaneous) KB of overhead memory currently swapped.

ovhd.mem.swapIn – (cumulative) KB of overhead memory swapped in for the current session.

ovhd.mem.swapOut – (cumulative) KB of overhead memory swapped out for the current session.

Expected values for some of the statistics:

vm.cpu.contention.mem – usually < 1%, anything greater indicates memory overcommit.

vm.cpu.contention.cpu – < 5% of incremental time during undercommit, < 50% of incremental time at normal levels of overcommit (vSphere is tuned to perform best when somewhat overcommitted).

When contention is < 5%, performance will be deterministic but the host is not fully used.

When contention is between 5% and 50%, the host is becoming fully used (maximum CPU throughput) but individual virtual machines might see less deterministic performance.

vm.numa.local – Expected to match guest.memory.mapped. Transient conditions such as NUMA rebalance can cause this to temporarily decrease, then return to normal as memory is migrated.

vm.numa.remote – Expected to be approximately zero in non-overcommitted scenarios.

guest.mem.mapped – Expected to equal configured guest memory; might be smaller if virtual machine has yet to access all its memory.

guest.mem.consumed – Expected to be approximately equal to configured guest memory; will be smaller if host memory is overcommitted.

guest.mem.swapped – Expected to be zero. Non-zero indicates non-graceful memory overcommit.

guest.mem.ballooned – Expected to be zero. Non-zero indicates graceful memory overcommit.

ovhd.mem.swapped – Expected to be zero. Non-zero indicates memory overcommit.

Equations for CPU and memory metrics:

session uptime = vm.cpu.used + vm.cpu.contention.cpu + vm.cpu.contention.mem + CPU idle time

configured memory size = guest.mem.mapped + guest.mem.swapped + (memory not yet touched)

configured memory size = `vm.numa.local` + `vm.numa.remote`
(another formula for arriving at the same statistic above)

`guest.mem.mapped` = `guest.mem.consumed` + `guest.mem.ballooned` + (other copy-on-write sources)

Comparison to esxtop

Individual reasons for lack of vCPU progress are available to vSphere administrators (using either `esxtop` or the vSphere API) but are hidden from the guest OS to preserve isolation between the virtual machine and the configuration of the infrastructure it runs upon. The guest OS sees only an aggregate metric.

`vm.cpu.used` is equivalent to the `esxtop` statistic %USED for a virtual machine.

`vm.cpu.contention.cpu` is equivalent to
(%RDY – %MLMTD) + %MLMTD + %CSTP + %WAIT + (%RUN – %USED)

(%RDY – %MLMTD) represents time the guest OS could not run due to host CPU overutilization. Note that %RDY includes %MLMTD, which is why it is subtracted before being added.

%MLMTD represents time the guest OS did not run due to administrator-configured resource limits. ESXi 6.0 and earlier did not add %MLMTD to this computation, but this is fixed in ESXi 6.5.

%CSTP represents time the guest OS could not run due to uneven vCPU progress.

%WAIT represents time the guest OS could not run due to hypervisor overheads.

(%RUN – %USED) corrects for any frequency scaling of the host CPU.

`vm.cpu.contention.mem` is equivalent to %SWPWT.

See https://communities.vmware.com/docs/DOC-9279 for details about `esxtop`.

Note on nominal CPU speed and CPU metrics

The `host.cpu.processorMHz` metric (in the host section) reports a nominal speed, and the virtual machine CPU metrics are normalized to the `processorMHz` metric. Actual processor speed might be higher or lower depending on host power management.

A virtual machine can see `vm.cpu.used` exceed wall clock time due to Turbo Boost, or can see `vm.cpu.used` lag wall clock time due to power saving modes used in conjunction with idle guests. Actual processor speed is not available to the guest OS, but is expected to be close to nominal clock speed when the guest OS is active. See http://www.vmware.com/files/pdf/techpaper/hpm-perf-vsphere55.pdf for more information about vSphere host power management.

Normalizing CPU metrics to nominal CPU speed allows the guest OS to avoid dependence on host power management settings.

Note on vm.cpu.contention.cpu

Using the Extended Guest Statistics discussed in this section, you can obtain a contention ratio by comparing contention time to actual time for a particular time interval. As contention time is reported as a sum across VCPUs, and wall time is reported for the entire virtual machine, the wall time must be scaled up by the number of VCPUs to normalize contention to a 0-100% range.

*Contention% = 100 * (contention_T2 – contention_T1) / (VCPUs * (time_T2 – time_T1))*

The `vm.cpu.contention.cpu` metric is similar to “stolen time” returned by `VMGuestLib_GetCpuStolenMs` (see Table 2-3), except “stolen time” excludes time the virtual machine did not run due to configured resource limits. Comparing this value to `esxtop` requires denormalizing the contention ratio, because `esxtop` reports a sum of percentages across VCPUs. So:

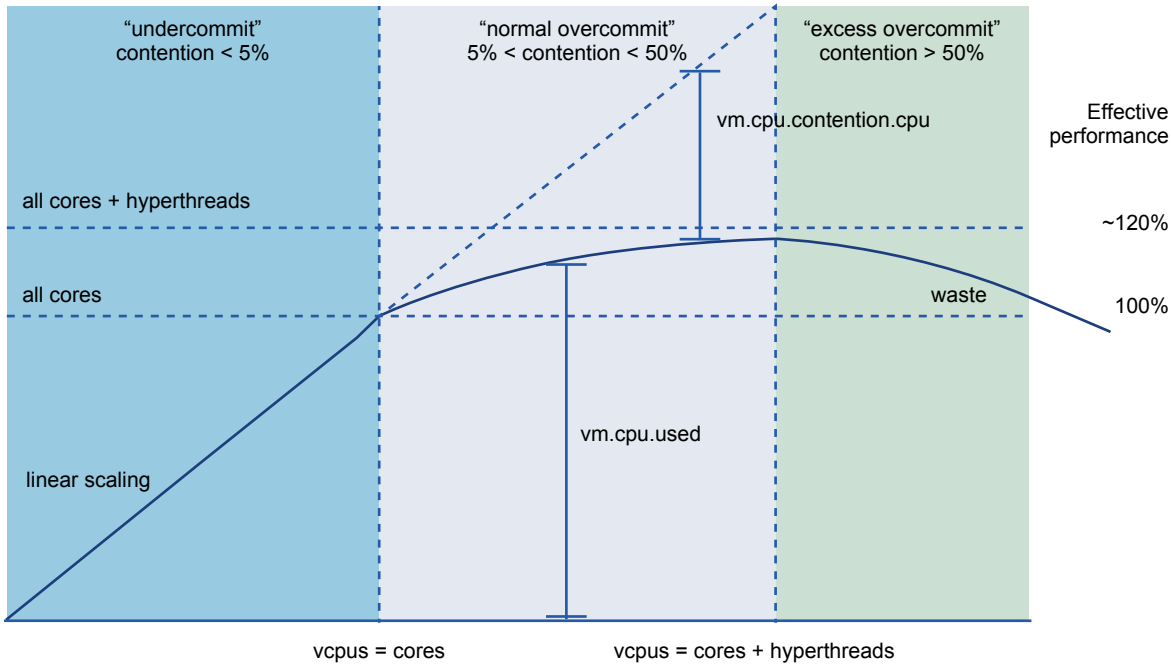
((%RDY – %MLMTD) + %MLMTD + %CSTP + %WAIT + (%RUN – %USED)) ~= *Contention% * VCPUs*

Due to sample aliasing where in-guest time samples and `esxtop` time samples do not occur simultaneously, instantaneous `esxtop` values will not match instantaneous in-guest statistics. Longer time samples or averaging values collected over time will produce more comparable results.

A contention value of < 5% is normal “undercommit” operating behavior, representing minor hypervisor overheads. A contention value > 50% is “excess overcommit” and indicates CPU resource starvation – the workload would benefit from additional CPUs or migrating virtual machines to different hosts. A contention value between 5% and 50% is “normal overcommit” and is more complicated. The goal of this metric is to allow direct measurement of the performance improvement that can be obtained by adding CPU resources.

The following figure illustrates these concepts.

Figure 3-1. CPU use across virtual machines



VMware best practices describe the available CPU capacity of an ESXi host as equal to the number of cores (not including hyperthreads). A 16 core host with 2.0GHz processors has 16 cores * 2000 MHz/core = 32000 MHz available compute capacity. When actual usage is below that calculated capacity, the hypervisor is considered “under committed” – the hypervisor is scaling linearly with load applied, and is wasting capacity.

As actual usage exceeds available compute capacity, the hypervisor begins utilizing hyperthreads for running virtual machines to keep performance degradation graceful. Maximum aggregate utilization occurs during this “normal overcommit” (between 5% and 50% contention) where each virtual machine sees somewhat degraded performance but overall system throughput still increases. In this “normal overcommit” region, adding load still improves overall efficiency, though at a declining rate. Eventually, all hyperthreads are fully used. Efficiency peaks and starts to degrade; this “excess overcommit” (>50% contention) indicates the workload would be more efficient if spread across more hosts for better throughput.

One specific scenario deserves special mention: the “monster VM” that attempts to give a single VM all available compute capacity. A VM configured to match the number of host cores (not including hyperthreads) will peak at the capacity of those cores (with < 5% contention) but at a performance about 20% lower than an equivalent physical machine utilizing all cores and hyperthreads. A VM configured to match the number of host threads (2x host cores) will peak at a performance level more analogous to a physical machine, but will show about 40% contention (the upper end of “normal overcommit”) running half the cores on hyperthreads. This contention metric indicates the load would run better on a larger host with additional cores, so it is technically “overcommitted” even though performance is better than a hypervisor running at full commit. This behavior is expected when attempting to run maximally sized virtual machines.

Storage Statistics

The following example shows some I/O statistics:

```
$ vmware-toolbox-cmd stat raw text vscsi scsi0:0
num.reads = 12605
num.writes = 1039
size.reads = 533612032
size.writes = 14279680
latency.reads = 1944173239
latency.writes = 102025122
```

`num.reads` – read commands.

`num.writes` – write commands.

`size.reads` – bytes read.

`size.writes` – bytes written.

`latency.reads` – microseconds of all read commands.

`latency.writes` – microseconds of all write commands.

The read IOPs over the last 10 seconds can be calculated as:

$$\frac{(\texttt{num.reads} \text{ @ } \textit{TimeNow}) - (\texttt{num.reads} \text{ @ } \textit{Time10sec})}{\textit{TimeNow} - \textit{Time10sec}}$$

The average latency of reads over the last ten seconds can be calculated as:

$$\frac{(\texttt{latency.reads} \text{ @ } \textit{TimeNow}) - (\texttt{latency.reads} \text{ @ } \textit{Time10sec})}{(\texttt{num.reads} \text{ @ } \textit{TimeNow}) - (\texttt{num.reads} \text{ @ } \textit{Time10sec})}$$

Network Statistics

Reservation and limit are supported on DVS (Distributed Virtual Switch) or “opaque” (NSX) switch types only; they are not supported on the default VSS switch type. Between `reservation` and `limit`, bandwidth is allocated on a share-based system, which is not meaningful to expose to a guest OS.

```
$ vmware-toolbox-cmd stat raw text vnet 00:0c:29:1e:23:f3
num.tx = 209
size.tx = 38137
num.rx = 10920
size.rx = 1312789
reservation = 0
limit = -1
```

`num.tx` – number of packets transmitted.

`size.tx` – bytes transmitted.

`num.rx` – number of packets received.

`size.rx` – bytes received.

`reservation` – guaranteed minimum bandwidth for this vNIC.

`limit` – maximum bandwidth allowed for this vNIC.

vSphere HA Application Monitoring

4

This chapter discusses the vSphere High Availability (HA) Application Monitoring and the following topics:

This chapter includes the following topics:

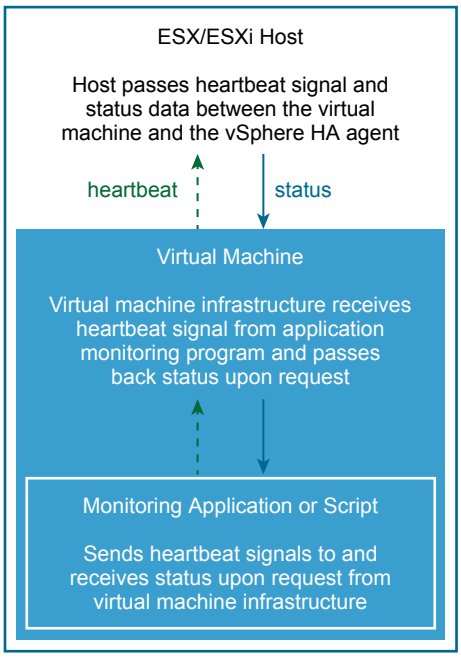
About vSphere HA

The vSphere High Availability (HA) feature for ESXi hosts in a cluster provides protection for a guest OS and its applications, by restarting the virtual machine if a guest OS or application failure occurs. The HA feature provides this reset capability using two different mechanisms:

- VM Monitoring – Guest OS heartbeats issued by the VMware Tools process.
- Application Monitoring – Heartbeats issued by a program that uses the HA Application Monitoring SDK to communicate with the VMware Tools process and the vSphere HA agent. This mechanism involves local monitoring by the program to avoid the overhead of sending messages to and from vCenter Server.

The following figure depicts the monitoring and reset capability of host and virtual machine.

Figure 4-1. Heartbeat and status signals



Additionally in vSphere 5.5 and later, the in-guest agent can set state to indicate it needs an immediate reset. This can be done without enabling heartbeats. The HA Application monitoring facility can reset the guest OS when ready to do so, if the in-guest agent has not changed state to say reset is no longer needed.

Using the HA Application Monitoring SDK, developers can write HA application monitoring programs in the C or C++ language. The HA Application Monitoring API is available with C language bindings only.

The application monitoring program sends an enable request to start the monitoring, possibly followed by a heartbeat signal. The vSphere infrastructure passes the signal up from your HA application monitoring program to the virtual machine, and then to the ESXi host. The HA application monitoring facility will reset the virtual machine if the application monitoring program stops sending a heartbeat signal, or requests a reset.

For more information about vSphere HA and application monitoring, see the *vSphere Availability* guide in the vSphere Documentation Center.

Prerequisites for HA Application Monitoring

Before you start working with the HA Application Monitoring SDK, verify that your vSphere application is running within a VMware cluster that has both the **High Availability** and **VM and Application Monitoring** options enabled.

You must install VMware Tools on the virtual machines where your HA monitoring applications are running.

The *vSphere Availability* guide contains information about how to set up a high availability (HA) cluster, and how to configure **VM and Application Monitoring**. With VMware's New Cluster Wizard, you can choose one of three monitoring options:

- **Disabled** – Neither VM Monitoring nor Application Monitoring.
- **VM Monitoring Only** – If you select this option, you will have the Guest OS monitoring discussed previously (the first mechanism).
- **VM and Application Monitoring** – If you select this option, you will also have the ability to employ Application Monitoring and the HA Application Monitoring SDK (the second mechanism).

For information about Web services interfaces for HA, see the *VMware vSphere API Reference Guide*, especially data objects `VirtualMachineRuntimeInfo` and `VirtualMachineRuntimeInfoDasProtectionState`.

Using the HA Application Monitoring APIs

You can use the HA Application Monitoring SDK to create a stand-alone application monitoring program, or to enhance an existing application or script. The purpose of your application monitoring program determines the API call sequence and the application behavior that you write to handle the response data.

For example, if your application monitoring program is tracking critical applications that are running in a guest OS, your application can intentionally stop sending heartbeat signals if any application-related process fails. The HA monitoring agent interprets the absence of heartbeats as a failure, and resets the virtual machine.

Alternatively, instead of not sending heartbeat signals, your application monitoring program can set the `needReset` flag using the `VMGuestAppMonitor_PostAppState` call. When the HA monitoring agent notices this flag, it will reset the virtual machine.

Most of the calls you make using the HA Application Monitoring APIs send information to the virtual infrastructure of the ESXi host, and the host relays the information to the HA monitoring agent. However, the `VMGuestAppMonitor_GetAppStatus` call is a two-way transaction that lets you request the virtual machine status from the HA monitoring agent.

Most HA Application Monitoring functions lack input parameters, because the calls are local. The vSphere infrastructure passes the heartbeat and status data to and from other levels of the cluster.

Call each function from your application monitoring program. The vSphere infrastructure (in the virtual machine where the application monitoring program is running) passes the function data up to the ESXi host. The local virtual machine sends all status responses to your application monitoring program, even though they are passed down from the HA monitoring agent.

HA Application Monitoring API Functions

The following calls are available to a vSphere HA application monitoring program:

Table 4-1. HA Application Monitoring API Calls

Call Name	Data Type Returned	Description
VMGuestAppMonitor_Enable	char	Requests the virtual machine infrastructure to monitor the calling application. The virtual machine infrastructure returns a value of VMGUESTAPPMONITORLIB_ERROR_SUCCESS, if monitoring was enabled. After your application monitoring program makes this call, your program must call VMGuestAppMonitor_MarkActive() at least once every 30 seconds or the virtual machine infrastructure will change the virtual machine's status to Red or Gray.
VMGuestAppMonitor_Disable	int	Requests the virtual machine infrastructure to stop monitoring the calling program. The virtual machine infrastructure returns a value of TRUE if monitoring was disabled.
VMGuestAppMonitor_IsEnabled	int	Returns the current recorded state of application monitoring. The virtual machine infrastructure returns a value of TRUE if monitoring is enabled.
VMGuestAppMonitor_MarkActive	char	Sends a request to mark the program as active. This function is also called the heartbeat because your program must call it at least once every 30 seconds while your application monitoring is enabled, or the virtual machine infrastructure will determine that the monitoring has failed.
VMGuestAppMonitor_PostAppState	int	Publish the application state that the guest OS wants delivered to vSphere HA. The application should monitor its environment and update its state accordingly. Heartbeat counting does not need to be enabled as a pre-condition, so the `enable()` call is not necessary. Returns 0 (zero) on success. The single `state` parameter passed to this call can be either: ■ OK – The guest's application agent declared state to be normal and no action is required. ■ `needReset` – The guest's application agent has requested an immediate reset. The guest can request this at any time.

Table 4-1. HA Application Monitoring API Calls (Continued)

Call Name	Data Type Returned	Description
VMGuestAppMonitor_GetAppStatus	char	Returns the current status recorded by the virtual machine infrastructure as ‘Green’, ‘Red’, or ‘Gray’. ■ Green. Virtual machine infrastructure acknowledges that the application is being monitored. ■ Red. Virtual machine infrastructure does not think the application is being monitored. The HA monitoring agent will initialize an asynchronous reset on the virtual machine if the status is Red. ■ Gray. Application should send VMGuestAppMonitor_Enable again, followed by VMGuestAppMonitor_MarkActive, because either application monitoring failed, or the virtual machine was vMotioned to a different location. Use the VMGuestAppMonitor_Free function to free the result. If this call returns a nonerror result that was not anticipated, it can mean that another program in the same virtual machine has called VMGuestAppMonitor_Disable or VMGuestAppMonitor_Enable. If your application is still running, call VMGuestAppMonitor_Enable again, followed by calls to VMGuestAppMonitor_MarkActive.
VMGuestAppMonitor_Free	char	Returns a pointer to the memory to be freed.

Code Sample for appmon.cpp

The HA Application Monitoring SDK includes a code sample called `appmon.cpp`. The sample is located in the `docs/samples` directory and defines the entry point for the console application. The `appmon.cpp` program includes interface code that your application monitoring program can send after receiving results from calls to VMGuestAppMonitor_Enable, VMGuestAppMonitor_MarkActive, and VMGuestAppMonitor_Disable.

Calling the APIs from Your Application

The following steps provide a possible API sequence of calls:

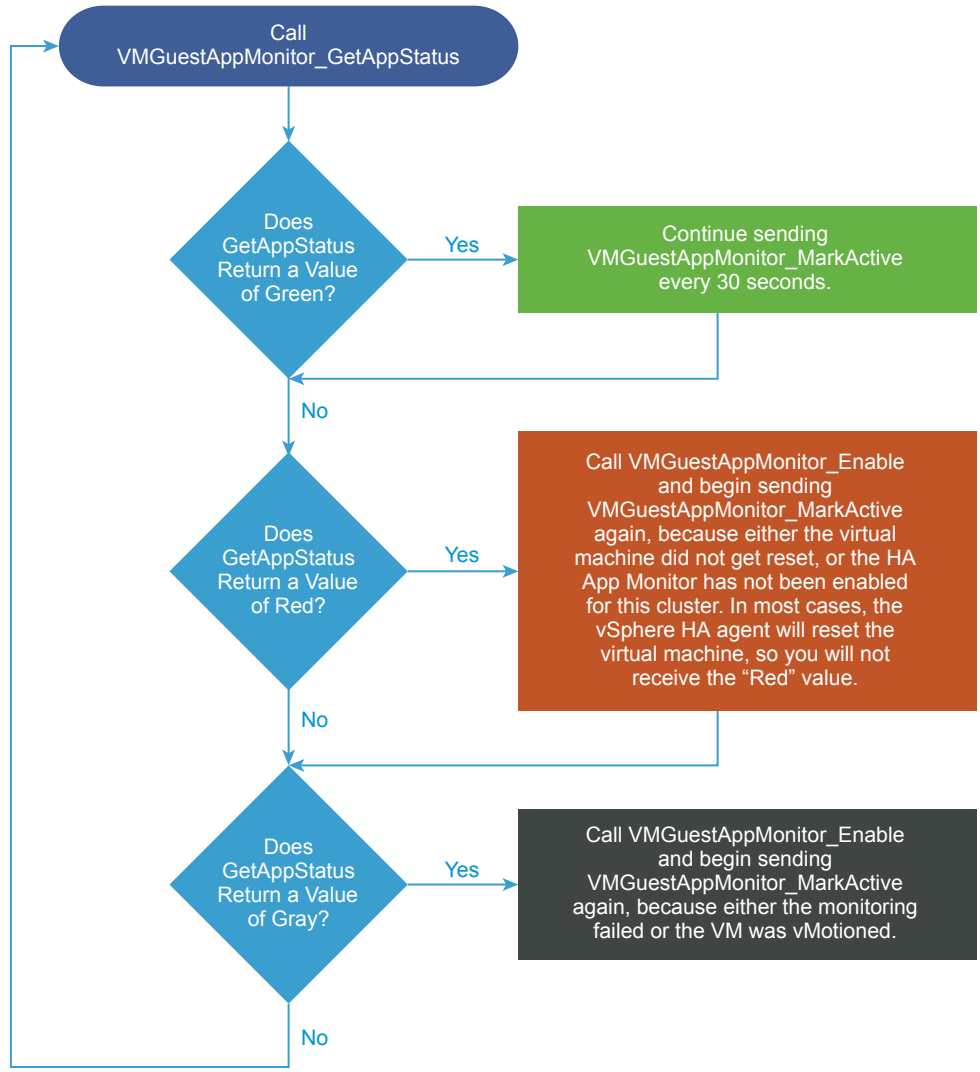
Procedure

1. Include `vmGuestAppMonitorLib.h` in the declarations for your C program.
2. To start the monitoring, notify the virtual machine that you are going to start sending a heartbeat signal by calling #unique_42_Connect_42_ID-3875-000006A9.
3. After you have called VMGuestAppMonitor_Enable, call #unique_42_Connect_42_ID-3875-000006C2 every 30 seconds or your virtual machine will be reset.
4. Send VMGuestAppMonitor_IsEnabled to make sure the virtual machine infrastructure received your requests correctly and has begun monitoring.

5 Periodically, call #unique_42_Connect_42_ID-3875-000006D8 to make sure the vSphere infrastructure is still receiving the heartbeat calls.

The status will be returned as *Green*, *Red*, or *Gray*. See Table 4-1 for a description of each status value. The figure below shows a possible coding flow for the GetAppStatus call.

Figure 4-2. Coding flow for VMGuestAppMonitor_GetAppStatus



6 After you call VMGuestAppMonitor_GetAppStatus, call the #unique_42_Connect_42_ID-3875-000006EA function to free the memory that was used to store the status.

If your application does not free the memory, it can use a large amount of storage very quickly because a new status is created every 30 seconds, when VMGuestAppMonitor_MarkActive is called.

7 Call #unique_42_Connect_42_ID-3875-000006B2 when you want the agent to stop monitoring.

HA Application Monitoring API Error Messages

The vSphere infrastructure can return errors in Table 4-2 as a result of HA Application Monitoring calls.

Table 4-2. HA Application Monitoring Error Codes

Error Message	Data Type	Code	Description
VMGUESTAPPMONITORLIB_ERROR_SUCCESS	int	0	Call completed successfully.
VMGUESTAPPMONITORLIB_ERROR_OTHER	char		Unknown error.
VMGUESTAPPMONITORLIB_ERROR_NOT_RUNNING_IN_VM	char		Calling application is not running within a virtual machine.
VMGUESTAPPMONITORLIB_ERROR_NOT_ENABLED	char		Monitoring is not enabled.
VMGUESTAPPMONITORLIB_ERROR_NOT_SUPPORTED	char		Monitoring is not supported.