Platform SDK Developer's Guide

LCA Hang-Up Detection Support

12/15/2025

Contents

LCA Hang-Up Detection Support

This page provides:

- an overview and list of requirements for the LCA Hang-Up Detection Support feature
- design details explaining how this feature works
- code examples showing how to implement LCA Hang-Up Detection Support in your applications

Introduction to LCA Hang-up Detection Support

Beginning with release 8.1, the Platform SDKs now allow user-developed application to include hang-up detection functionality.

The Genesys Management Layer relies on Local Control Agent (LCA) to monitor and control applications. An open connection between LCA and Genesys applications is typically used to determine which applications are running or stopped. However, if an application that has stopped responding still has a connection to LCA then it could appear to be running correctly - preventing Management Layer from switching over to a backup application or taking other actions to restore functionality.

Hang-up detection allows Local Control Agent (LCA) to detect unresponsive Genesys applications by checking for regular heartbeat messages. When an unresponsive application is found, pre-configured actions can be taken - including triggering alarms or restarting the application.

Note: Hang-up detection functionality has been available in the Genesys Management Layer since release 8.0.1. For more information, refer to the Framework 8.0 Management Layer User's Guide. For details about related configuration options, refer to the Framework 8.0 Configuration Options Reference Manual.

Two levels of hang-up detection are available: implicit and explicit.

Implicit Hang-up Detection

The easiest form of hang-up detection to implement is implicit hang-up detection.

In this scenario, application status is monitored through the connection between your application and LCA. This functionality can be extended by adding a requirement that your application periodically interacts with LCA (either responding to ping request or sending its own heart-beat messages) as a necessary condition of application liveliness.

This simple form of hang-up detection can be implemented internally by using the `LocalControlAgentProtocol` to connect to LCA. In this case, existing applications only need to be rebuilt with a version of `LocalControlAgentProtocol` that supports hang-up detection functionality - no coding changes are required - and given the appropriate configuration options in Genesys Management Layer.

Explicit Hang-up Detection

Explicit hang-up detection offers more robust protection from applications that may become unresponsive, but is also more complex.

The periodic interaction that is monitored by implicit hang-up detection only confirms that your application can interact with LCA. In most cases this means that the application is able to communicate with other apps and that the thread responsible for communicating with LCA is still active. However, multi-threaded applications may contain other threads that are blocked or have stopped responding without interrupting communication with LCA. Explicit hang-up detection allows you to determine when only part of your application hangs-up by monitoring individual threads in the application.

In addition to allowing your application to register (or unregister) individual threads to be monitored, explicit hang-up detection also allows your application to stop or delay the monitoring process. Threads that execute synchronous functions (which can block thread execution for some extended periods) or other features that prevent accurate monitoring should take advantage of this feature.

Feature Overview

- To maintain backwards compatibility, hang-up detection must be explicitly enabled in the application configuration.
- Implicit hang-up detection can be used for applications that do not require complex monitoring functionality. No code changes are required, just rebuild your application using the new version of `LocalControlAgentProtocol`.
- Explicit hang-up detection requires minimal application participation - enabling monitoring, registering and unregistering execution threads, and providing heartbeats. Most hang-up detection functionality is implemented within the Management Layer component, while all timing information (such as maximum allowed period between heartbeats) is configured through Genesys Management Layer.

System Requirements

Genesys Management Layer:

- Release 8.0.1 or later

Platform SDK for .NET:

- Management SDK protocol release 8.1 or later
- .NET Framework 3.5, 4.0 or 4.5
- Visual Studio 2008 (required for .NET project files)

Platform SDK for Java:

- Management SDK protocol release 8.1 or later

- Java SE 5, 6 or 7

Design Details

This section provides an overview of the main classes and interfaces used to add thread monitoring functionality for Explicit hang-up detection. Before using the classes and methods described here, be sure that you have implemented basic LCA Integration in your application using `LocalControlAgentProtocol`.

Although the details of thread monitoring implementation are slightly differently for Java and .NET, the basic idea is the same: to create and update a thread monitoring table that LCA can use to confirm the status of your application.

Note that for implicit hang-up detection you are only required to rebuild your application and make adjustments to the configuration options in Genesys Management Layer; the details described below are not required for simple application monitoring.

Thread Monitoring Table

The new thread monitoring functions described below allow `LocalControlAgentProtocol` to create and maintain a thread monitoring table within the application. This table tracks basic thread status.

Sample Thread Monitoring Table

OS Thread ID	Logical Thread ID	Thread Class	Heartbeat Counter	Flags
0	«main»	1	444345	active
1	«pool_1»	2	354354	suspend
2	«pool_2»	2	432432	deleted
3	«pool_3»	2	434323	active
4	«DB_store»	3	31212	active
....	....	....	....	....

Each row corresponds to a monitored thread. Columns of the table are:

- OS Thread ID—The OS-specific thread ID, used for thread identification during monitoring. OS thread ID is not passed by application but is received directly from system.
- Logical Thread ID – Application logical thread ID (or logical name, in Java). Used for logging and thread identification.
- Thread Class—Thread class integer. This value is only meaningful within the scope of the application; threads with the same thread class value in a different application can have different roles. Examples of thread classes might be the main loop thread, pool threads, or special threads (such as external authentication threads in ConfigServer).
- Heartbeat Counter—Cumulative counter of `Heartbeat()` calls made by the corresponding thread. Incrementing this value is the main way to indicate that the thread is still alive.

NOTE: This value is initialized with a random value when the thread is registered for monitoring. This prevents incorrect hang-up detection if threads are created and terminated with high frequency, leading to repeating OS thread IDs.

- Flag—Special flags.
  - Suspended/Resumed—Corresponds to the state of thread monitoring.
  - Deleted—Used internally to notify LCA that a thread was unregistered from monitoring.

.NET Implementation

ThreadMonitoring Class

The `ThreadMonitoring` class is defined in the `Genesyslab.Diagnostics` namespace of Genesyslab.Core.dll. This class contains the following public static methods:

- `Register(int threadClass, string threadLogicId)`—enables monitoring for this thread
- `Unregister()`—removes this thread from monitoring
- `Heartbeat()`—increases heartbeat counter for this thread (indicating that thread is still alive)
- `SuspendMonitoring()`—suspend monitoring for this thread
- `ResumeMonitoring()`—resumes monitoring for this thread

Note: Each method should be called from within the thread that is being monitored.

When a thread is registered for monitoring, the following parameters are included:

- `threadClass`—Any positive integer that represents the type of thread, allowing you to specify different monitoring settings for groups of threads within an application.
- `threadLogicId`—A logical, descriptive thread ID that is independent from thread ID provided by OS. This value is used for thread identification within LCA and for logging purposes. This ID should be unique within the application.

PerformanceCounter Constants

The following String constants (names) are defined in the `ThreadMonitoring` class:

```
public const string CategoryName = "Genesyslab PSDK .NET";
public const string HeartbeatCounterName = "Thread Heartbeat";
public const string StateCounterName = "Thread State";
public const string ProcessIdCounterName = "ProcessId";
public const string OsThreadIdCounterName = "OsThreadId";
```

The Platform SDK thread monitoring functionality uses these constants to manage PerformanceCounter values. In addition to these custom performance counters, you can also use standard ones, such as those defined in `Thread` category: "% Processor Time", "% User Time", etc.

See MSDN<ref>MSDN PerformanceCounter Class (http://msdn.microsoft.com/en-us/library/system.diagnostics.performancecounter.aspx)</ref> for details about performance counters.

Note: Use of PerformanceCounters is optional, and is not required for LCA hang-up detection functionality.

Java Implementation

ThreadHeartbeatCounter class

The `ThreadHeartbeatCounter` class is defined in the `com.genesyslab.platform.commons.threading` package, located within commons.jar. This class is designed as a JMX<ref>JMX: Java Management Extensions (http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/)</ref> MBean and implements the public ThreadHeartbeatCounterMBean interface which is accessible through Java management framework.

There is no public constructor for the `ThreadHeartbeatCounter` class; each thread that you want to monitor should create its own instance with following static method:

```
public static ThreadHeartbeatCounter createThreadHeartbeatCounter(
            String threadLogicalName,
            int threadClass);
```

When a thread is registered for monitoring, the following parameters are included:

- `threadLogicalName`—A logical, descriptive thread name that is used to identify the thread within LCA and for logging purposes. This name should be unique within the application.
- `threadClass`—Any positive integer that represents the type of thread, allowing you to specify different monitoring settings for groups of threads within an application.

One key difference from thread monitoring using .NET is the need to create a monitoring object instance. The lifecycle of this object, including `MBeanServer` registration, is supported by the parent class `PSDKMBeanBase` and is shown in the five steps below:

1. Start monitoring a thread:

```
ThreadHeartbeatCounter monitor =
  ThreadHeartbeatCounter.createThreadHeartbeatCounter(
                threadId, threadClass);
monitor.initialize();
```

2. Notify LCA that thread is still alive (increase heartbeat counter):

```
monitor.alive();
```

3. Suspend monitoring of this thread:

```
monitor.setActive(false);
```

4. Resume monitoring of this thread:

```
monitor.setActive(true);
```

5. Finish monitoring and unregister this thread:

```
monitor.unregister();
```

Note: Each of these methods must be called from within the thread that is being monitored.

Once a `ThreadHeartbeatCounter` object is unregistered, that instance cannot be reused. To begin monitoring that thread again (or any other) you first need to create a new instance of the thread monitoring object.

ThreadHeartbeatCounterMBean interface

The `ThreadHeartbeatCounterMBean` interface is intended to present an open API to the JMX MBean. This interface contains the following publicly accessible methods:

```
    public long getThreadSystemId();
    public String getLogicalName();
    public int getThreadClass();
    public void setThreadClass(int newThreadClass);
    public int getHeartbeatCounter();
    public void setActive(boolean isActive);
    public boolean isActive();
```

These methods are "MBean client-side" methods and are used by LCA protocol to get actual information about the thread for the monitoring table. They also allow users to change the thread class and suspend or resume thread monitoring (using `setActive(false/true)`) of a particular thread at application runtime.

References

<references />