Performance Sentry Proxy SDK

May 24, 2012

Author: Mark Fuini

Reviewer: Phil Henninge

1. Revision History 1
2. Overview 1
3. API Architecture 2
4. SDK Components 2
4.1. Performance Sentry 2
4.2. Performance Sentry Proxy 2
4.2.1. Performance Sentry Proxy Installation 3
4.2.1. Performance Sentry Proxy Configuration 5
4.2.2. Performance Data Management Interface 6
4.3. SDK Client Samples 7
4.3.1. Web Service Reference 8
4.3.2. Configuration 9
4.3.3. Making the request 10
4.3.4. Examining the data 12
4.3.5. PowerShell Sample Code 13
4.3.6. Building a UI Application 16

1. Revision History

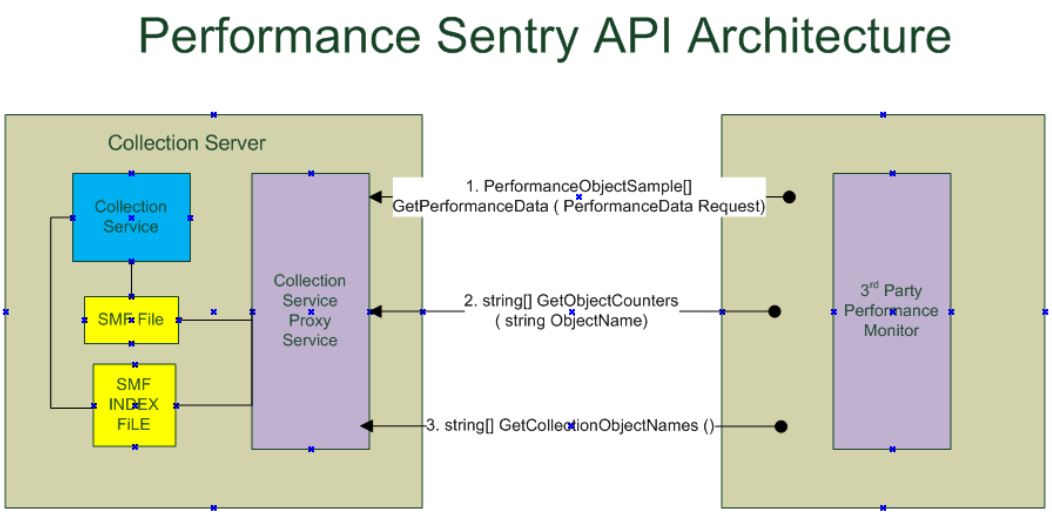
Author	Date	Version	Description
Mark Fuini	11-22-2011	V0.1	Initial Version without Powershell Sample Code
Mark Fuini	12-21-2011	v1.0	Updated Sample Code with the ClientEx application. Updated Collection Service Proxy Install procedure.
Phil Henninge	12-23-2011	V1.0	Review and Edit
Mark Fuini	12-26-2011	V1.0	Updated from Review Comments
Phil Henninge	05-04-2012	V1.0	Review and Edit

2. Overview

Thank you for your interest in Demand Technology Software's Performance Sentry product.

This document describes the Performance Sentry SDK. The SDK will allow 3rd Party applications to retrieve historical performance data from a particular machine running the Performance Sentry Collection Service. Third party applications will be able to retrieve data collected by the service as needed for troubleshooting instead of having to process the entire SMF data collection file in ad-hoc situations, or as a daily scheduled event.

3. API Architecture Overview



The Performance Sentry Application Programming Interface (API) is used by third party software to communicate with the Performance Sentry Proxy Service. The Proxy Service receives requests for information through the API then uses the SMF Index file to reference the performance data in the SMF file and return the requested data to the API calling program.

4. SDK Components

4.1. Performance Sentry

Performance Sentry Collection Service for Windows should be installed on the machine where performance data is to be collected. Performance Sentry writes Windows performance data to the SMF files. Refer to the Performance Sentry User Manual for more information about Performance Sentry installation and operation.

4.2. Performance Sentry Proxy

The Performance Sentry Proxy is a new component which provides 3rd party applications access to the performance data collected by the Performance Sentry Collection Service on a particular machine. This component is installed as a Windows service on each machine where Performance Sentry Collection Service is running. This component is a Microsoft .NET Service which uses Windows Communication Foundation (WCF) for communication.

4.2.1. Performance Sentry Proxy Installation

The Performance Sentry Proxy service requires version 4.0 of the .NET Framework and at minimum, version 4.0.0.5 of the Performance Sentry Collection Service.

The Performance Sentry Proxy is installed with the Performance Sentry Collection Service beginning with version 4.0.0.13.

The Performance Service Proxy must be installed with the Microsoft .NET Installer Tool program called installutil.exe. Developers of Windows Services in Microsoft.Net will be familiar with this process of installing .NET services.

Start by unzipping the PerformanceSentryProxy.zip file to a new folder, preferably:

C:\Program Files\NTSMF40

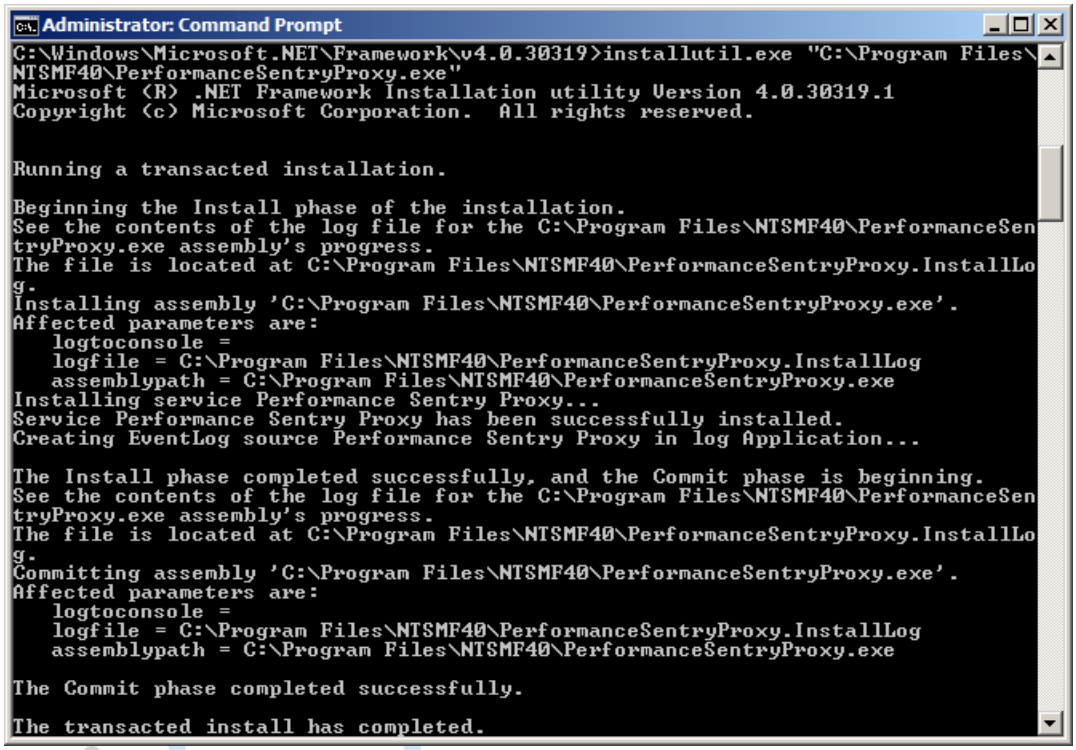
After unzipping the PerformanceSentryProxy.zip file, navigate to the Microsoft .NET 4.0 directory on your system:

C:\Windows\Microsoft.NET\Framework\v4.0.xxxxxx

and execute the following command:

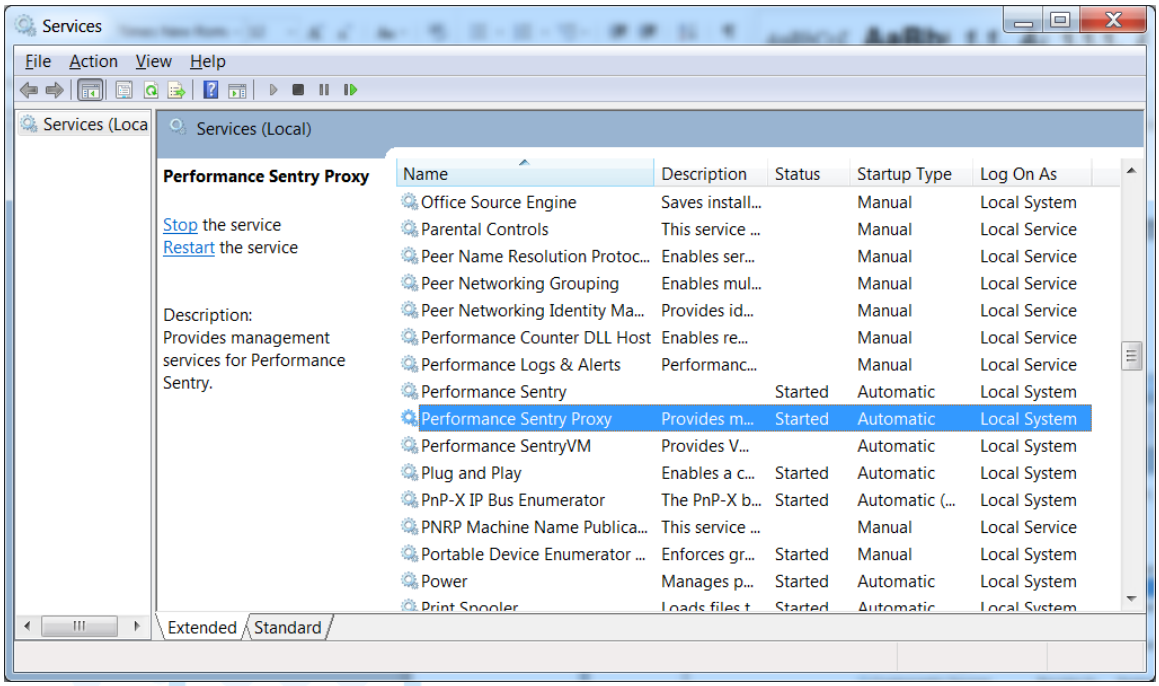
installutil.exe "C:\Program Files\NTSMF\PerformanceSentryProxy.exe"

Note: You can unzip the files contained in PerformanceSentryProxy.zip to any folder you choose and change the installutil.exe command accordingly.



When the command successfully finishes, the message "The transacted install has completed" will be displayed as in the screen capture above.

To verify the server has started, look at the windows services for the Performance Sentry



4.2.1. Performance Sentry Proxy Configuration

The Performance Sentry Proxy hosts a performance data manager WCF service that can be configured by editing the **PerformanceSentryProxy.exe.config** file in the installation directory. There are two endpoints configured by default. One uses HTTP and the other uses NetTcp. The addresses that need to be specified when connecting to the Performance Sentry Proxy from another machine are shown in the baseAddresses section. This configuration can be modified as needed.

```
<services>
      <!-- Performance Data -->
      <service behaviorConfiguration="PerformanceDataManager.Behavior"
        name="DTS.CollectionServiceProxy.PerformanceDataManager">

        <endpoint address="" binding="netTcpBinding"
bindingConfiguration="NetTcpStreamingBindingEndpoint"
          name="NetTcpStreamingBindingEndpoint"
          contract="DTS.CollectionServiceProxy.IPerformanceDataMgmt" />

        <endpoint address="" binding="basicHttpBinding"
bindingConfiguration="basicHttpStreamingBinding"
          contract="DTS.CollectionServiceProxy.IPerformanceDataMgmt" />

        <endpoint address="mex" binding="mexTcpBinding"
name="MexTcpBindingEndpoint"
          contract="IMetadataExchange" />

        <host>
          <baseAddresses>
            <add baseAddress="net.tcp://localhost:5558/DTSPSP" />
```

```
            <add baseAddress="http://localhost:5557/DTSPSP" />
          </baseAddresses>
        </host>

      </service>
```

The bindings section of the configuration is also important. This contains the bindings for each endpoint. By default, a streamed binding is used. In general, the **maxReceivedMessageSize** should be about 100kb. The message size can be increased if more historical data is necessary.

```
    <bindings>
      <basicHttpBinding>

       <binding name="basicHttpStreamingBinding"
       maxReceivedMessageSize="100000"
          transferMode="Streamed" />
      </basicHttpBinding>
      <netTcpBinding>

        <binding name="NetTcpStreamingBindingEndpoint" closeTimeout="00:01:00"
          openTimeout="00:01:00" receiveTimeout="00:10:00"
sendTimeout="00:01:00"
          transactionFlow="false" transferMode="Streamed"
          transactionProtocol="OleTransactions"
          hostNameComparisonMode="StrongWildcard" listenBacklog="10"
          maxBufferPoolSize="524288"
          maxBufferSize="65536" maxConnections="10"
          maxReceivedMessageSize="100000">
          <readerQuotas maxDepth="32" maxStringContentLength="8192"
          maxArrayLength="16384"
            maxBytesPerRead="4096" maxNameTableCharCount="16384" />
          <reliableSession ordered="true" inactivityTimeout="00:10:00"
            enabled="false" />
          <security mode="Transport">
             <transport clientCredentialType="Windows"
             protectionLevel="EncryptAndSign" />
            <message clientCredentialType="Windows" />
          </security>
        </binding>
…
      </netTcpBinding>

    </bindings>
```

If these settings are changed, the proxy service must be restarted.

4.2.2. Performance Data Management Interface

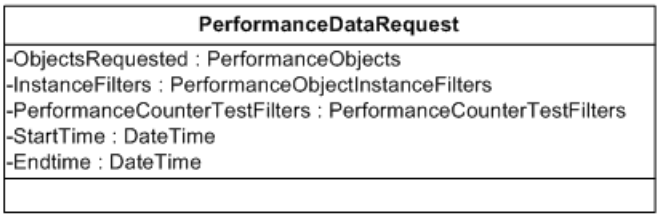
Historical performance data can be retrieved from a particular machine by using the **IPerformanceDataMgmt** interface of the Performance Sentry Proxy.

The performance data management interface is shown below.

```
    [ServiceContract]
    public interface IPerformanceDataMgmt
    {
```

```
        // Returns performance data
        [OperationContract]
        PerformanceData GetPerformanceData(PerformanceDataRequest Request);

        // Returns the ordered counters from the discovery record
        [OperationContract]
        string[] GetCounterNames(string Object);

        // Returns objects from the discovery record
        [OperationContract]
        string[] GetCollectionObjectNames();
    }
```

Performance data is retrieved by passing a **PerformanceDataRequest** object to the proxy through the **GetPerformanceData** method. The **PerformanceDataRequest** object requires a start time, end time, and list of object data to be returned over the requested time period.



Instance filters can be specified to restrict the data being returned to specific instances of the objects requested. Counter test filters can also be specified to return only instances of objects for which the specified counter meets the test. These filters can be used to reduce the amount of data returned to the caller. These mechanisms will be further explained in the sample code examples.

In the event that the caller needs to know which counters are being collected for a particular object, the **GetCounterNames** method can be used.

To find out the objects which have been collected, on the machine the caller can utilize the **GetCollectionObjectNames**.

4.3. SDK Client Samples

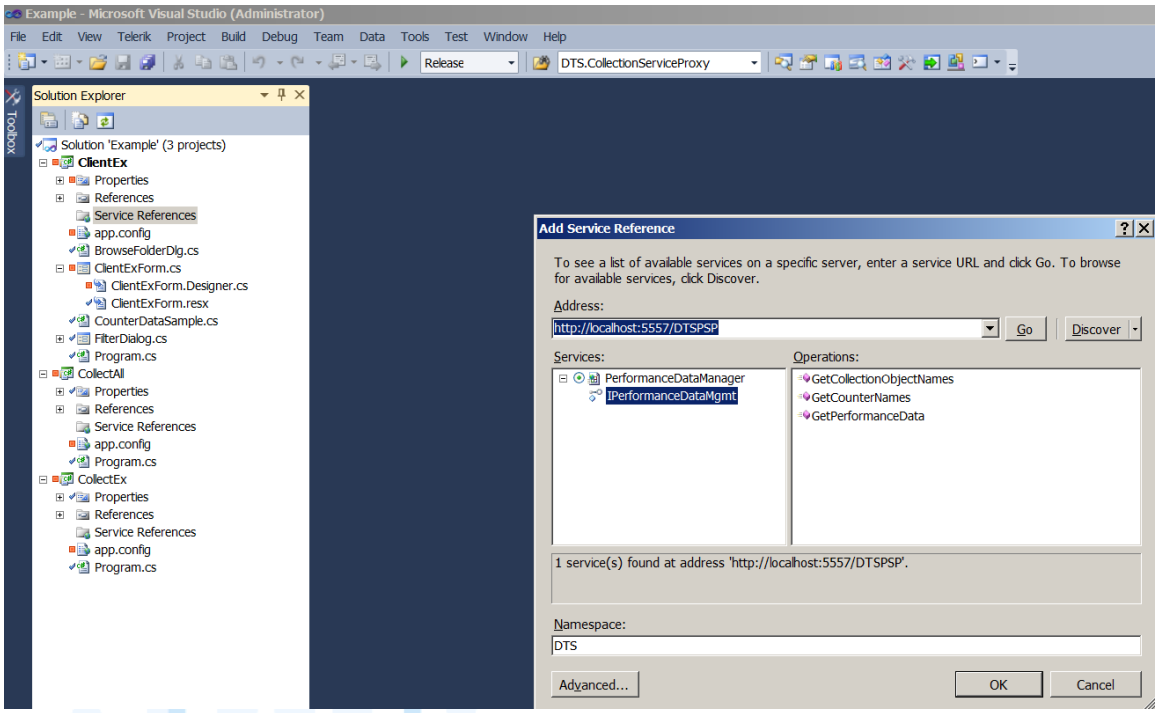
The SDK provides sample code to develop client applications. The program examples were written in in C# using Visual Studio 2010. Examples using Microsoft Powershell v2 are also provided. Other platforms which support WebServices can also be utilized depending on customer requirements.

The table below lists the sample projects included in the SDK and their descriptions:

Project Name	Description	APIs Exercised	Platform
CollectEx	Sample code to collect specific objects for the last 30 minutes based upon using specified counters, all counters, and instance and counter filters.	GetPerformanceData, GetCounterNames	C#, Powershell v2
CollectAll	Sample code to collect all counters of one specific object for the last 30 minutes. Object name is passed for an argument.	GetPerformanceData, GetCounterNames	C#, Powershell v2
ListObjects	Sample to show how to make call to the Get CollectionObjects API	GetCollectionObject Names	Powershell v2
ClientEx	Sample to show how to develop a client UI application for performance data retrieval.	GetCollectionObjectNames, GetPerformanceData, GetCounterNames	C#

4.3.1. Web Service Reference

A web service reference is created in a Visual Studio project by right clicking on the **Service References** item and selecting **Add Reference**. This has already been done in the sample project. When adding a service reference, the URI of the web service is specified. In this case, the URI will be the same location as what is configured in the **Performance Sentry Proxy** for the HTTP binding. In the example code, the DTS namespace was specified when creating the service reference.



4.3.2. Configuration

When the WebService reference is added, entries for the configuration will be added to the app.config file.

There will be a configuration for the binding. This is where options such as the **maxReceivedMessageSize** can be modified.

```
  <system.serviceModel>
       <bindings>
           <basicHttpBinding>
              <binding name="BasicHttpBinding_IPerformanceDataMgmt"
              closeTimeout="00:01:00"
                    openTimeout="00:01:00" receiveTimeout="00:10:00"
                    sendTimeout="00:01:00"
                    allowCookies="false" bypassProxyOnLocal="false"
                    hostNameComparisonMode="StrongWildcard"
                    maxBufferSize="65536" maxBufferPoolSize="524288"
                    maxReceivedMessageSize="65536"
                    messageEncoding="Text" textEncoding="utf-8" transferMode="Buffered"
                    useDefaultWebProxy="true">
                    <readerQuotas maxDepth="32" maxStringContentLength="8192"
maxArrayLength="16384" maxBytesPerRead="4096" maxNameTableCharCount="16384" />
…
                </binding>
            </basicHttpBinding>
```

There is also a client configuration address with the end points of the added web service. The address of the endpoint is specified according to the location of the **Performance**

Sentry Proxy. If your application is monitoring more than one collection service, the proxy can be created dynamically so it does not rely upon this endpoint base address.

```
   <client>
…
        <endpoint address="http://localhost:5557/DTSPSP"  binding="basicHttpBinding"
                 bindingConfiguration="BasicHttpBinding_IPerformanceDataMgmt"
                contract="DTS.IPerformanceDataMgmt"
                name="BasicHttpBinding_IPerformanceDataMgmt" />

  </client>
```

4.3.3. Making the request

In order to request data, a data request object is created. The data request has an object list indicating which objects to retrieve historical data. These objects are instantiated and assigned a particular name. **Note: Any object, counter, or instance names specified in the data request object are case-sensitive.**

```
// Set up the object we would like to collect
DTS.PerformanceObject perfObject = new DTS.PerformanceObject();
perfObject.Name = "Process";
```

Counters are assigned to the object for data that the user wants to retrieve. It is important to only request data which will be used in order to minimize bandwidth for the request.

```
perfObject.Counters = new PerformanceCounter[3];

DTS.PerformanceCounter ProcessorTime = new DTS.PerformanceCounter();
ProcessorTime.Name = "% Processor Time";
DTS.PerformanceCounter UserTime = new DTS.PerformanceCounter();
UserTime.Name = "% User Time";
DTS.PerformanceCounter IdProcess = new DTS.PerformanceCounter();
IdProcess.Name = "ID Process";

perfObject.Counters[0] = ProcessorTime;
perfObject.Counters[1] = UserTime;
perfObject.Counters[2] = IdProcess;
```

Once the object is created, it can be added to an object list which will be assigned to the data request object later in the process.

```
// Object list
ArrayList ObjectList = new ArrayList();
ObjectList.Add(perfObject);
```

If all of the counters of a particular object are needed, then the client can call the function **GetCounterNames** for a particular object. This will return a list of the counters being collected for the object. This list can then be used to build the list of counters to be collected.

```
// Get the counters for the thread object and retrieve them all
```

```
string[] CounterNames = client.GetCounterNames("Memory");
if (CounterNames != null)
{
      // Get the thread object
      DTS.PerformanceObject threadObject = new DTS.PerformanceObject();
      threadObject.Name = "Memory";
      threadObject.Counters = new PerformanceCounter[CounterNames.Length];
      for (int i = 0; i < CounterNames.Length; i++)
      {
           PerformanceCounter pc = new PerformanceCounter();
           pc.Name = CounterNames[i];
           threadObject.Counters[i] = pc;
      }
      ObjectList.Add(threadObject);
}

// Add the object to the list of objects for the request
PerformanceObject[] perfObjects = new PerformanceObject[ObjectList.Count];
       for (int i = 0; i < ObjectList.Count; i++)
            perfObjects[i] = (PerformanceObject) ObjectList[i];
```

Object instance filters can be used to reduce bandwidth to restrict collection to only specific instances of an object. In the example below, the Process object data is only returned for the instances specified.

```
      DTS.PerformanceObjectInstanceFilter filter = new
DTS.PerformanceObjectInstanceFilter();
      filter.ObjectName = "Process";
      filter.InstanceNames = new string[] { "svchost", "DmPerfss" };
```

The user may not want to receive all instances for the object if the time interval is large. To restrict data even further, a counter filter was created to filter on a counter of an object. The threshold can be a test whether the counter is less than, less than or equal, greater than, greater than or equal, or just equal to a value.

In the following example only process instances with a value of "% Processor Time" > 1% are returned.

```
// Set up a counter test filter for % Processor Time, instances
// which don't meet the threshold will be not returned
DTS.PerformanceCounterTestFilter counterTest = new
DTS.PerformanceCounterTestFilter();
counterTest.ObjectName = "Process";
counterTest.CounterName = "% Processor Time";
counterTest.Value = 1.0;
counterTest.Type = DTS.PerformanceCounterTestType.GT;
```

When the performance data objects and counters are configured, then the performance data request can be configured. The objects requested are specified, along with end time, start time, and any applicable filters.

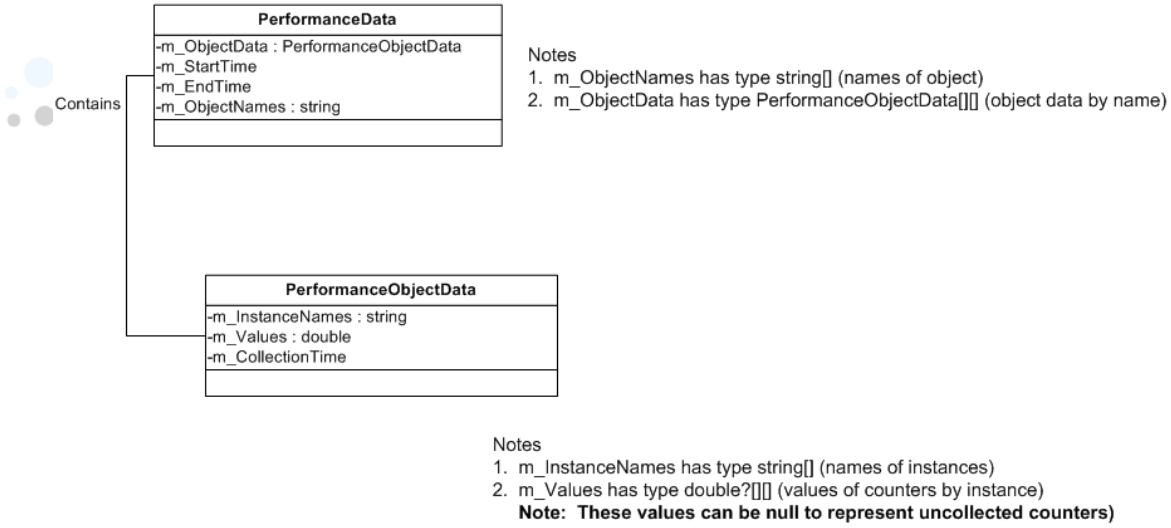
```
// Set up the request
```

```
        DTS.PerformanceDataRequest request = new DTS.PerformanceDataRequest();
        request.EndTime = DateTime.Now;
        request.StartTime = request.EndTime - new TimeSpan(0, 30, 0);
        request.ObjectsRequested = perfObjects;
        request.InstanceFilters = new DTS.PerformanceObjectInstanceFilter[] {
filter };
        request.PerformanceCounterTestFilters = new
DTS.PerformanceCounterTestFilter[] { counterTest };
```

4.3.4. Examining the data

To request the data, the following call is made passing the 'request' object. The 'request' is of type **PerformanceDataRequest**.

```
        // Get the performance data
        DTS.PerformanceData data = client.GetPerformanceData(request);
        if (data==null)
        throw new Exception("Could not get performance data.");
```



The **PerformanceData** class contains an **ObjectNames** field which is an array of strings with the names of the objects to fulfill the request. If the first object request is 'Process', then ObjectNames[0] will be 'Process' The **ObjectData** field is the 'jagged' array of **PerformanceObjectData** instantiations for each object. If the first object requested is 'Process', then the PerformanceObjectData for the process will be in the array in the location of ObjectData[0][].

The **PerformanceObjectData** class is utilized to hold the counter values of each instance returned for a particular requested object at a particular collection time. The names of the instances returned are in the **InstanceNames**[] string array. The name of the first instance of an object at a particular collection time indexed by the InstanceNames[0] value. Uninstanced objects will have a blank instance name.

The actual counter values are in a jagged array called **Values**. The counter values are indexed in the same order as they are requested. For the first instance of a particular object, the Values would be in the Values[0][] array. **These values must be null-checked since uncollected counter values are represented by null instead of 0.0 or some distinct numerical value. See the highlighted text in the example below.**

The following code constructs a message containing the object instances and values:

```
        // Data is returned in the ObjectData array for each object name
        for (int i = 0; i < data.ObjectNames.Length; i++)
        {
          // Instances of the object are stored in the ObjectData array sorted by Time
          for (int instance=0; instance < data.ObjectData[i].Length; instance++)
          {
             // Get the sample for the instance of the ith object
             PerformanceObjectData sample = data.ObjectData[i][instance];

    string message = data.ObjectNames[i] + "," +
sample.CollectionTime.ToString();

             // Each object has a number of instances
            for (int j=0; j < sample.InstanceNames.Length; j++)
            {
                 message =  message + "\r\n" + sample.InstanceNames[j];
                // Each instance has a number of values ordered by the request order
                for (int k = 0; k < sample.Values[j].Length; k++)
                {
                     message = message + ",";

                    if (sample.Values[j][k] != null)
                        message += sample.Values[j][k];
                }
            }
        }
```


4.3.5. PowerShell Sample Code

The Performance Sentry Proxy webservice can also be accessed via PowerShell v2. Starting with PowerShell v2, there is a new New-WebServiceProxy cmdlet that automatically generates a proxy to the specified web service. Before executing the sample "collect" script, enter the following command into PowerShell to generate the web service proxy. The –uri parameter should be the configured URI of the http endpoint for Collect Service Proxy.

```
$proxy = New-WebServiceProxy -Uri 'http://localhost:5557/DTSPSP' -namespace DTS
```

This proxy has to be generated one time for each PowerShell session. If the proxy is generated a second time with the same DTS namespace, then PowerShell will generate a superfluous casting error to the same type due to a Microsoft bug. For this reason, it is recommended that the $proxy variable be used for the life of the PowerShell session.

Once the $proxy variable is created and the script is loaded, then the collect command can be typed at the command line in PowerShell repeatedly.

The example script shows the process object with 3 counters, instance filter, and counter filter with data being retrieved for the last 30 minutes.

```
# example collect script to collect data from the collection service proxy

# execute this before the script so that the DTS proxy variable is set up:

# $proxy = New-WebServiceProxy -Uri 'http://localhost:5557/DTSPSP' -namespace DTS

# this can only be done once per powershell session since powershell has a bug
# that will create a superfluous casting error if the proxy is created twice

function collectex
{

# create object to collect
$object = New-Object DTS.PerformanceObject
$object.Name = 'Process'

#create the filter
$filter = New-Object DTS.PerformanceObjectInstanceFilter
$filter.ObjectName = 'Process'
$filter.InstanceNames = @('svchost','DmPerfss')
```

A counter filter test can be created in Powershell just as in C#.

```
#counters
$countertestfilter = New-Object DTS.PerformanceCounterTestFilter
$countertestfilter.ObjectName = 'Process'
$countertestfilter.counterName = '% Processor Time'
$countertestfilter.ThresholdValue = 0.1
$countertestfilter.ThresholdValueSpecified = $true
```

In order to specify the type of counter test, the new enum object is created. To assign the value, use the integer value of the enum. Using the integer value is a disadvantage over a standard enumerated type but PowerShell does not seem to recognize the actual enumerated type values to do the assignment. Lastly, the 'Type Specified' flag must be set. This flag must be set or the proxy does not transmit the value to the web server.

```
$testtype = New-Object DTS.PerformanceCounterTestType
$testtype.value__ = 1  # enums are converted to int in ascending order starting with
zero LT,GT,LTE,GTE,EQ
$countertestfilter.Type = $testtype
$countertestfilter.TypeSpecified = $true
```

The rest of the example is similar to the C# example above with some modifications for Powershell syntax.

```
# filter the counters
$counter = New-Object DTS.PerformanceCounter
$counter.Name = '% Processor Time'

$counter2 = New-Object DTS.PerformanceCounter
$counter2.Name = '% User Time'
```

```
$counter3 = New-Object DTS.PerformanceCounter
$counter3.Name = 'ID Process'

$object.counters = @($counter,$counter2,$counter3)

# set up the memory object to collect all of the counters
$mobject = New-Object DTS.PerformanceObject
$mobject.Name = 'Memory'

$mcounters = $proxy.GetCounterNames('Memory')
if ($mcounters -eq $null)
{
    Write-Host 'Could not get the counternames for the Memory object'
}
else
{
    $mobject.counters = @()
    foreach ($countername in $mcounters)
    {
        $mcounter = New-Object DTS.PerformanceCounter
        $mcounter.Name = $countername;
        $mobject.counters += @($mcounter)
    }

    # set up the time
    $endtime = [System.DateTime]::Now
    $duration = New-Object System.TimeSpan(0,0,30,0)
    $starttime = $endtime - $duration

    # create the request
    $request = New-Object DTS.PerformanceDataRequest
    $request.ObjectsRequested = @($object,$mobject)
    $request.InstanceFilters = @($filter)
    $request.PerformanceCounterTestFilters = @($countertestfilter);
    $request.StartTime = $starttime
    $request.EndTime = $endtime
    $request.StartTimeSpecified = $true
    $request.EndTimeSpecified = $true

    Write-Host 'Collecting from ' $uri ' Object(s) ' $request.ObjectsRequested '
FilteredInstances(s) ' $request.InstanceFilters
    Write-Host 'From ' $starttime ' To ' $endtime

    # get the data
    $data = $proxy.GetPerformanceData($request)

    # Data is returned in the ObjectData array for each object name
    for ($i = 0; $i -lt $data.ObjectNames.Length; $i++)
    {
        # samples of the object are stored in the ObjectData array sorted by Time
        for ($s = 0; $s -lt $data.ObjectData[$i].Length; $s++)
        {
            # Get the sample of the ith object
            $sample = $data.ObjectData[$i][$s]

            $message = $data.ObjectNames[$i] + ',' + $sample.CollectionTime

            # Each sample has a number of instances, for each instance output the
counter values
```

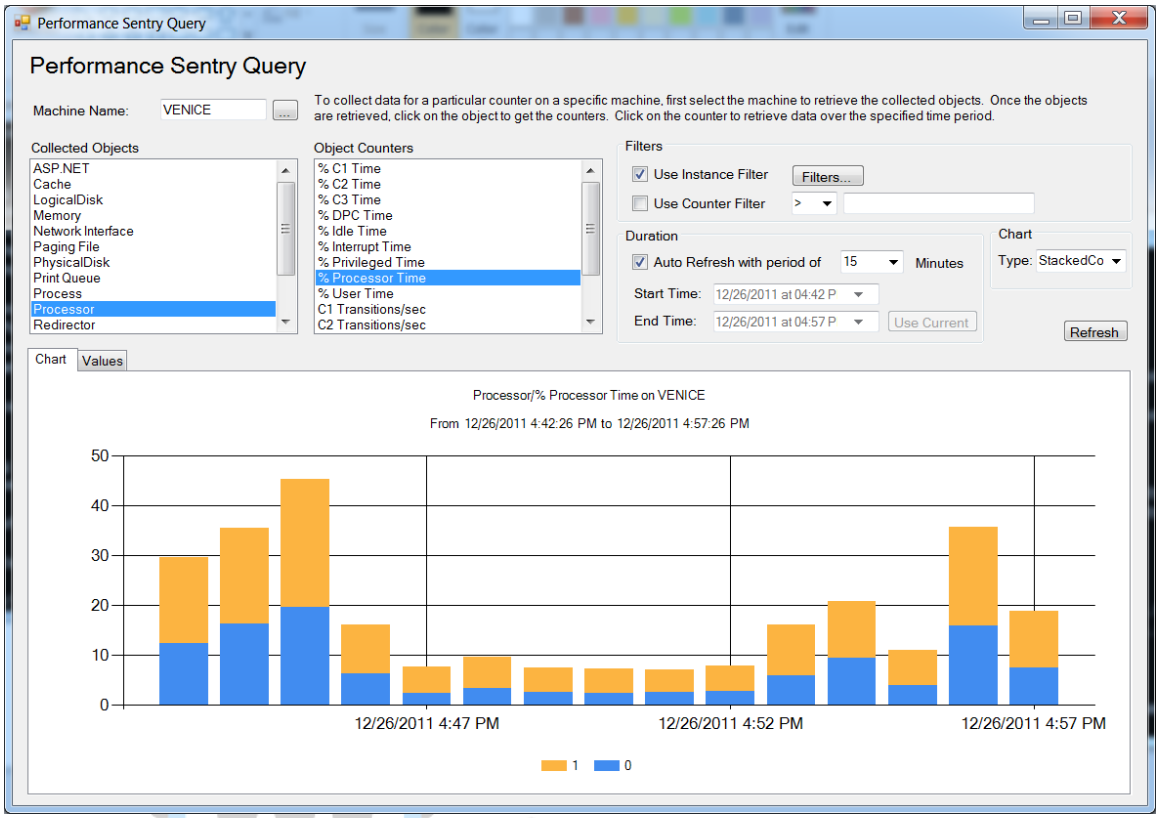
```
            for ($j=0; $j -lt $sample.InstanceNames.Length; $j++)
            {

                $message =  $message + "`r`n" + $sample.InstanceNames[$j]

                # Each instance has a number of values ordered by the request counter
order
                for ($k = 0; $k -lt $sample.Values[$j].Length; $k++)
                {
                    $message = $message + ','

                    if ($sample.Values[$j][$k] -ne $null)
                    {
                        $message = $message + $sample.Values[$j][$k]
                    }
                }
            }

           Write-Host $message
        }
    }
}

$uri = $null
$message = $null
$object = $null
$counter = $null
$filter = $null
$result = $null
$request = $null
$proxy = $null

}
```

4.3.6. Building a UI Application

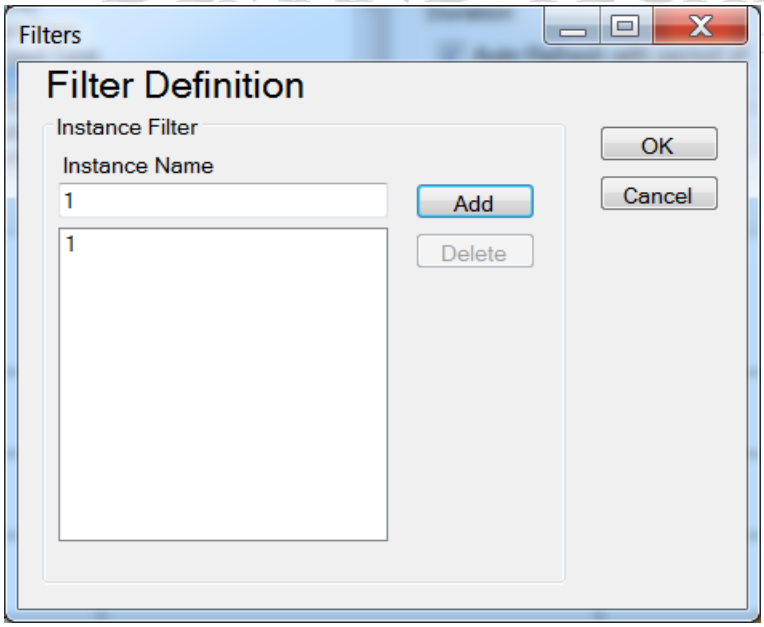
One of the ways developers can utilize the API is by building applications which retrieve historical data in near real-time for a particular machine. The SDK includes a sample application called ClientEx to demonstrate how such an application might work.

The application allows the user to browse network machines to retrieve recent or historical data. Once a machine is selected by the user, the collected objects on the machine are retrieved by calling the **GetCollectedObjects** method on the **Performance Sentry Proxy**.

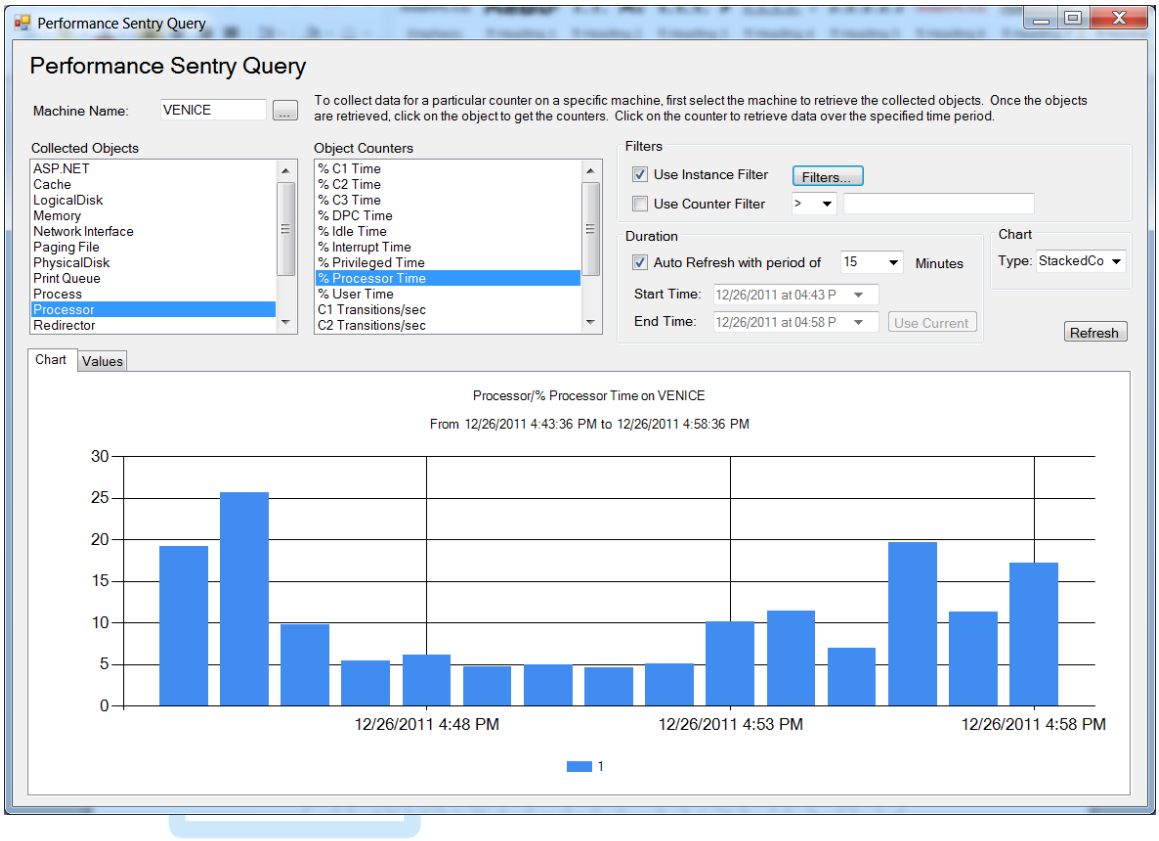
Once the object is the selected, the program retrieves the available counters for the object on the selected machine by calling the **GetCounterNames** method. When a counter name is selected, a data request for that counter is created for the selected duration. The duration of data to be returned can be manually selected or set to auto refresh. The data request is then passed to the **GetPerformanceData** method on the proxy to the selected machine. The results are shown on the chart and values tab.



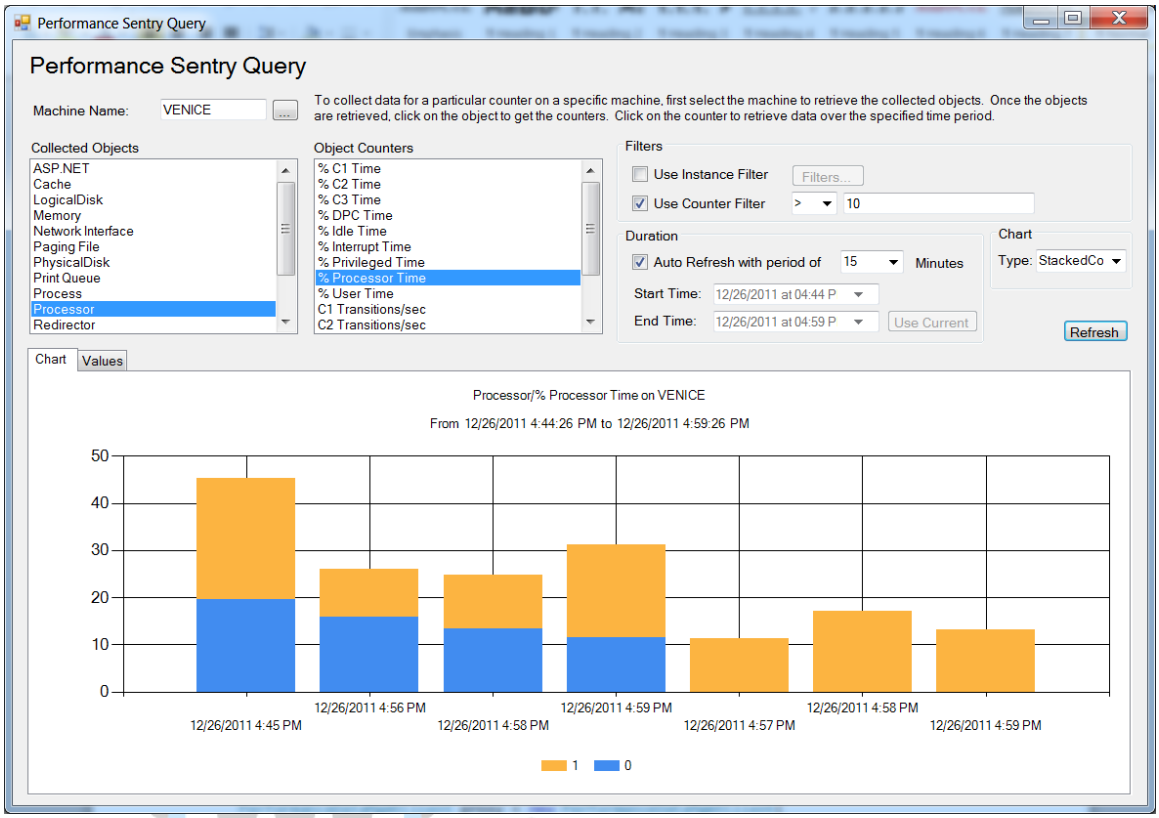
The sample demonstrates the API instance and counter test filters. The 'use instance filter' button enables the instance filter and allows the user to select the 'Filters' button. The 'Filters' button launches a dialog box where the user can enter instance names.



After pressing ‘OK’, you can see the results of the filter after you press refresh. Only data



The counter test filter allows the user to enter a value and select an operation to perform with it against the actual counter value of any object instances in the specified time period. If the counter value does not meet the test for a particular instance of the performance object, then that instance of the object is not returned. The following example shows data returned if the ‘% Processor Time’ is > 10% over the specified duration.



It is important to note, that since the user can decide which machine to monitor, the proxy object is created dynamically. The format of the base address is read from the configuration file with the $MachineName replaced with the machine name selected by the user to retrieve data as shown below.

```
string baseAddress =
Properties.Settings.Default.PerfSentryDataMgmtAddress.Replace("$MachineName",
MachineName);

PerformanceDataMgmtClient proxy = new PerformanceDataMgmtClient(
                Properties.Settings.Default.PerfSentryDataMgmtBinding,
                baseAddress);
```

This example demonstrates how to utilize the functions of the Performance Sentry Proxy to build a 'recent history' data collection application with the Performance Sentry SDK.