QuickCounters.net

Scott Colestock
Trace Ventures, LLC
blog: www.traceofthought.net
scott@traceventures.net

Agenda

- What is QuickCounters ?
- Windows Performance Counter background
- .Net Classes for Performance Counters
- Getting Up & Running w/QuickCounters
  - Demos…
- Support for BizTalk, WCF, WF

What is QuickCounters ?

- A shared source library (MS-CL) for rapidly adding instrumentation to service entry points
  - General .Net Components
  - Web Services
  - BizTalk Orchestrations
  - .Net Remoting Interfaces
  - Enterprise Service / COM+
  - MSMQ Queue-Reading Services

What is QuickCounters ?

- What kind of instrumentation? Specifically, request-level metrics for:
  - Requests Started
  - Requests Executing
  - Requests Completed
  - Requests Failed
  - Request Execution Time
  - Requests/Hour
  - Requests/Min
  - Requests/Sec

What is the benefit ?

- Describe your requests in a simple xml format
- Include code snippet in each request implementation:

```
void SampleRequest()
{
   RequestType someRequest = RequestType.Attach("MyApplication","someRequest");
   someRequest.BeginRequest();

   try
   {
      // Do useful work...
      someRequest.SetComplete();
   }
   catch
   {
      someRequest.SetAbort();
      throw;
   }
```

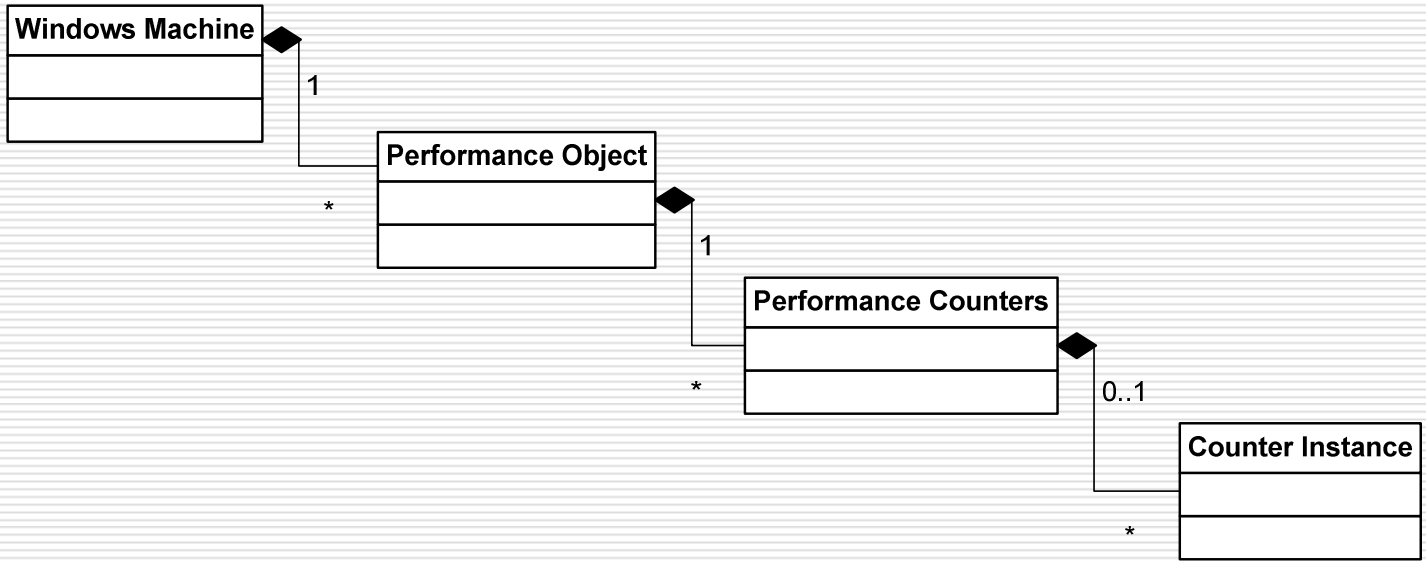
- Run your app, and view metrics in real time within Performance Monitor:

Color	Scale	Counter	Instance	Parent	Object	Computer
	1.000	Request Execution Time (msec)	---	---	QuickCountersUnitTest:...	\\TRACESOLAR
	1.000	Requests Completed	---	---	QuickCountersUnitTest:...	\\TRACESOLAR
	1.000	Requests Executing	---	---	QuickCountersUnitTest:...	\\TRACESOLAR
	1.000	Requests Failed	---	---	QuickCountersUnitTest:...	\\TRACESOLAR
	1.000	Requests Started	---	---	QuickCountersUnitTest:...	\\TRACESOLAR
	1.000	Requests/Hour	---	---	QuickCountersUnitTest:...	\\TRACESOLAR
	1.000	Requests/Min	---	---	QuickCountersUnitTest:...	\\TRACESOLAR
	1.000	Requests/Sec	---	---	QuickCountersUnitTest:...	\\TRACESOLAR

Performance Counter Background

- Perf Counters have always been in "Windows NT" lineage…
- Provide info as to how well an application, service, or driver is performing
  - Request level metrics
  - Other detailed counts or timings
  - Occasionally state information
- Operating system, hardware elements, and most commercial Windows services expose counters

Performance Counter Background

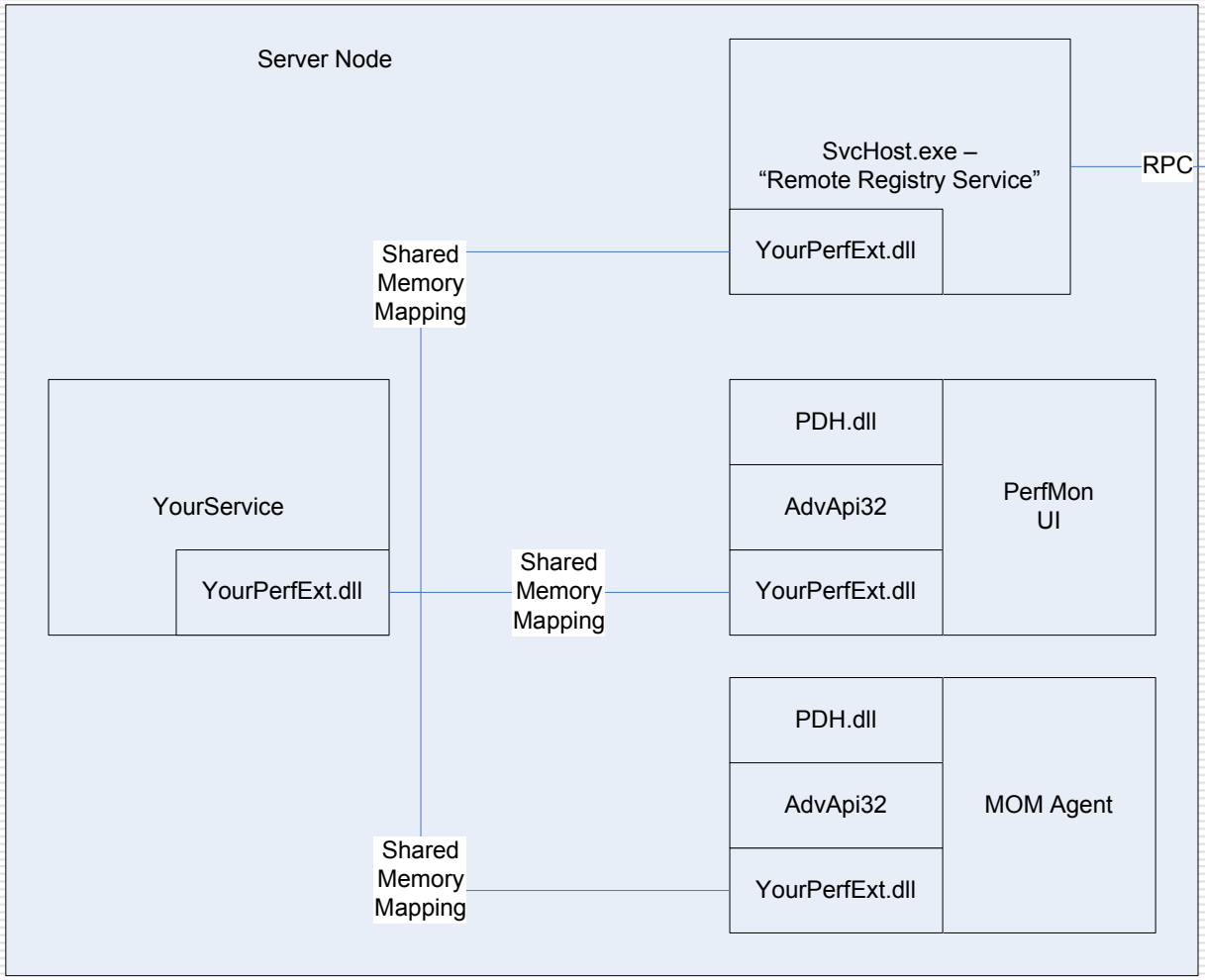


Performance Counter Background

- Perf counters give you the ability to obtain real-time metrics
  - No need to sift a log file
  - Can sample as needed for statistical or historical analysis
- Perf counters can be viewed remotely
- Consumable by management tools
- Perf counters have an extremely low overhead to both expose and consume…

Performance Counter Background

- Why such low overhead?



- YourPerfExt.dll maintains a set of data structures for performance data (objects, counters, instances) – in conformance with requirements imposed by Collect PerformanceData function it must implement.
- Your service simply writes data at will, though API of choice. YourPerfExt.dll puts data in the data structures.
- Data structures visible to your service (for writing) and other processes (for reading) via shared memory.

.Net Perf Counter Implementation

- Writing Perf Counter Extension Dlls in C/C++ was not for the faint of heart – easy to get wrong.
  - Powerful mechanism, seldom used by non-commercial software until recently
- .Net Libraries implement a “generic” perf counter extension Dll for all managed apps: “perfcounter.dll”
- Full set of managed classes for:
  - Installing performance counters
  - Writing values
  - Reading values

.Net Perf Counter Implementation

- System.Diagnostics namespace
- PerformanceCounterCategory
  - All about interrogating what performance objects are available (local/remote)
  - Can create/delete performance objects on local machine
    - Do with installer instead!
  - Can read all counters for the performance object
  - Name doesn't align…

.Net Perf Counter Implementation

- PerformanceCounter
  - Construct with particular performance object and counter names
    - Optionally instance name, machine name
  - Allows for reading (local/remote) and writing (local)
  - Methods for incrementing/decrementing
    - When counter is a rate, division by time sample length is automatic
- PerformanceCounterInstaller
  - Used in an Installer-derived class that you author
  - You configure the containing assembly as a “Custom Action” for your MSI
  - Handles installation of all of your counters when you describe name and type for each
  - Essentially a complicated registry update routine!

.Net Perf Counter Implementation

- Excellent abstractions provided
- Far easier than C/C++
- But…when I want request level metrics:
  - Requests Started
  - Requests Executing
  - Requests Completed
  - Requests Failed
  - Request Execution Time
  - Requests/Hour
  - Requests/Min
  - Requests/Sec
- There is still a lot of code to write!
  - PerformanceCounter class has to be instantiated and managed appropriately for all of these
  - Installation has to be managed
- (Do I need all those counters ?)
  - See Sql Server, ASP.NET, MSMQ, BizTalk, etc.

Hence QuickCounters

- Designed to raise the level of abstraction from the individual counter up to the “Request”
  - General .Net component method
  - Web Service operations
  - BizTalk Orchestrations
  - .Net Remoting interface methods
  - Enterprise Service / COM+ methods
  - MSMQ Queue-Reading Services
- Designed to handle manipulation of all performance counters automatically, as well as installation tasks
- Designed to provide additional services to particular runtime environments
  - BizTalk
  - WCF
  - WF

QuickCounters Step 1

- Install the QuickCounters MSI
  - http://shurl.org/quickcounters (CodePlex)
  - QuickCounters assembly installed in GAC
  - "Program Files" installdir will contain:
    - Viewer application
    - Unit test
    - QuickCounters assembly
  - Have your Visual Studio projects reference QuickCounters assembly from installation directory

QuickCounters Step 2

- Create an Xml file that describes the components and requests you wish to instrument.
  - RequestTypes = Performance objects (aka categories)
  - Each have eight counters…
  - Each component has counter for host process uptime

```
<InstrumentedApplication>
   <Name>NameOfYourApp</Name>
   <Description>Description of your app</Description>
   <Component>
      <Name>SomeAppComponent</Name>
      <Description>SomeAppComponent  Description</Description>
      <RequestTypes>
         <RequestType>
            <Name>SomeRequest</Name>
            <Description>SomeRequest Description</Description>
         </RequestType>
         <RequestType>
          …
         </RequestType>
      </RequestTypes>
   </Component>
   <Component>
    …
   </Component>
</InstrumentedApplication>
```

QuickCounters Step 3

- "Install" your xml file to create counters
  - Use Viewer application or…
  - installutil /quickctrconfig=YourCounters.xml QuickCounters.net.dll
  - Or configure QuickCounters.net.dll as a custom action in your MSI
    - For both install and uninstall
    - With /quickctrconfig=YourCounters.xml in the CustomActionData

QuickCounters Step 4

- Add appropriate code to your “request” implementations
  - RequestType instance should be created via “Attach” factory method.
    - Don’t allow multiple threads to use a single instance of a RequestType
    - Underlying “PerformanceCounter” instances are all cached across “Attach” calls

```
void SampleRequest()
{
   RequestType someRequest = RequestType.Attach("MyApplication","someRequest");
   someRequest.BeginRequest();

   try
   {
      // Do useful work...
      someRequest.SetComplete();
   }
   catch
   {
      someRequest.SetAbort();
      throw;
   }

}
```

Demo Recap

- Recap
  - We installed QuickCounters
  - We created InstrumentedApplication xml, and then “installed” it
  - We referenced QuickCounters assembly
  - We created a RequestType instance using Attach factory method
  - Called BeginRequest/SetComplete/ SetAbort
- Don’t lose sight…
  - We treated each key press as a request
  - Your requests will be:
    - General .Net component methods
    - Web Service operations
    - BizTalk Orchestrations
    - .Net Remoting interface methods
    - Enterprise Service / COM+ methods
    - MSMQ Queue-Reading (per message)
  - You should have a unique RequestType instance for each request! (“Attach” with component and request name)
    - Use the instance for the duration of the request.

A few implementation details

- "Per Second" counter is a standard "rate" counter (RateOfCountsPerSecond32)
- "Per Minute" and "Per Hour"
  - Implemented because many requests aren't measured well at per/second
  - Implemented by adding a timestamp to an array when we SetComplete/SetAbort
  - On timer interval (5 sec)
    - BinarySearch to find point in array representing values 1 minute (or 1 hour) ago
    - Prune the array before that (RemoveRange)
    - "Per Min"/"Per Hour" equal to the count of what is left in the array
- Execution time uses QueryPerformanceFrequency & QueryPerformanceCounter for high-resolution timings…unless the RequestType instance is serialized

BizTalk Support

- BizTalk brings several interesting complexities…
- Can create RequestType instance as the first step in the orchestration for orch-wide metrics
  - Can also create/use different RequestType instances for sub-portions of the orch
- Orchestrations will often begin on one node of a BizTalk group
  - …but continue (and/or complete) execution on the other nodes after dehydrate/rehydrate
- How do we ensure “Requests Executing”, “Requests Completed”, etc. all remain correct if BeginRequest and SetComplete will happen on different machines?

BizTalk Support

- RequestType is a fully serializable class – state will be included in dehydrated orchestrations
- "Attach" method has override that will accept orchestrationId
  - Used to listen for dehydration, suspension, and completion events
  - Requests Executing decremented…
- Serialization constructor will check for presence of orchesrationId
  - Increment Requests Executing
- Requests Completed/Requests Failed and the "per Second/Minute/Hour" metrics are "credited" to the server that completes the request

WCF Support

- WCF has powerful extensibility mechanisms – including for interception
- By implementing an IDispatchMessageInspector derivation, we can hook all methods of a WCF service implementation
- This means all Attach, BeginRequest, SetAbort calls can be made *for you*
  - "Attach" assumes component = service name, and request = method name

WCF Support

- WCF developer simply adds a [InstrumentedService] attribute to service declaration or…
- Define a behavior extension in app.config, pointing to QuickCounters.ServiceModel.Extensions.Configuration.InstrumentedServiceElement as the implementation
  - Add the behavior to your serviceBehaviors
- <InstrumentedApplication> xml can be auto-generated for you based on your wsdl

Future Directions

- Support for Windows Workflow, using interception model to avoid manual BeginRequest/SetComplete/SetAbort calls
- Support for System Center Operations Manager 2007
  - Auto-generate the initial portion of management pack xml

Contributors

- Myself
- Dave Comfort
- John Thom
- Tomas Restrepo
- You ?

Thanks! Questions ?

QuickCounters: http://shurl.org/quickcounters

Scott Colestock
Trace Ventures, LLC
blog: www.traceofthought.net
scott@traceventures.net