CLR Instrumentation Engine & Intercept Extension

This document introduces the CLR Instrumentation Engine and Intercept Extension to the OpenTelemetry .NET SIG. The main purpose is to give the high level design

CLR Instrumentation Engine

The CLR Instrumentation Engine (CLRIE) is a cooperation profiler that allows running multiple profiling extensions in the same process. It was built to satisfy Application Insights, IntelliTrace, and Production Breakpoints scenarios and enable cooperation of Microsoft auto-enabled instrumentation extensions with the profilers built by partners.

CLR Instrumentation Engine provides the multiplexing functionality to overcome the CLR limitation. It also simplifies some basic scenarios providing higher level interfaces. It simplifies IL rewrite by providing IL code tree abstraction and coordinating it between multiple extensions.

CLR Instrumentation Engine provides a pluggable mechanism of enabling additional extensions. Extensions may be enabled at configuration time as well as in runtime. Extensions enabled in runtime are the subject to all the limitations CLR imposes. For instance, an extension loaded in runtime cannot disable code optimization if it wasn't disabled by any pre-configured extensions.

Design

All the design notes can be found here.

Intercept Extension

Demo

After CLR Instrumentation Engine was enabled, this is how to configure the monitoring. We want to report every call of the method `SmtpClient.Send` as a client span.

First - import another NuGet package:

```
<package
      id="Microsoft.ApplicationInsights.Agent.Intercept"
      version="2.0.5" />
```

Second - call the method Decorate and pass the following method information - assembly name, module name and full method name.

```
Decorator.InitializeExtension();
Functions.Decorate("System", "System.dll",
"System.Net.Mail.SmtpClient.Send",
    OnBegin, OnEnd, OnException, false);
```

You also need to pass three callbacks:

```
public static object OnBegin(object thisObj, object arg1)
public static object OnEnd(object context, object returnValue,
object thisObj, object arg1)
public static void OnException(object context, object exception,
object thisObj, object arg1)
```

The call to Decorate will do the magic. It finds the method you specified and using the Runtime Instrumentation Agent inserts those callbacks into the beginning, end and in the global `try{}catch` statement of that method. This magic is only allowed when CLR Instrumentation Engine is enabled for the process.

In the callbacks implementation you can access method arguments and exchange the context:

```
public static object OnBegin(object thisObj, object arg1)
{
    // start the operation
    var operation = new
TelemetryClient().StartOperation<DependencyTelemetry>("Send");
    operation.Telemetry.Type = "Smtp";
    operation.Telemetry.Target = ((SmtpClient)thisObj).Host;
    if (arg1 != null)
    {
        operation.Telemetry.Data = ((MailMessage)arg1).Subject;
    }
    // save the operation in the local context
    return operation;
}
```

OnEnd and OnException callbacks:

```
public static void OnException(object context, object exception,
object thisObj, object arg1)
{
    // mark operation as failed and stop it. Getting the operation
from the context
    var operation =
(IOperationHolder<DependencyTelemetry>)context;
    operation.Telemetry.Success = false;
    operation.Telemetry.ResultCode = exception.GetType().Name;
    new TelemetryClient().StopOperation(operation);
}
```

Notice the runtime arguments passed to the original method are used in those callbacks to collect information. Argument called `thisObj` is an instance of `SmtpClient` that made a call and `arg1` is a `MailMessage` that was passed as an argument.

Design

There are quite a few interesting design choices implemented in Intercept extension. Here are some design choices made:

Callbacks

```
protected string CustomMethod(DateTime arg1, Object arg2)
{
    object retVal = null;
    object context = PrefixCode(this, arg1, arg2);


    try
    {
        #region Custom code
        // Custom code: ---------start---------
        // Custom code: Custom code without return instruction.
        string stackTopValue = "some string is already on the stack.";
        // Custom code: ----------end----------
        #endregion //Custom code
        retVal = stackTopValue;
        retVal = PostfixCode(context, retVal, this, arg1, arg2);
    }
    catch (Exception exc)
    {
        OnException(context, exc, this, arg1, arg2);
        throw;
```

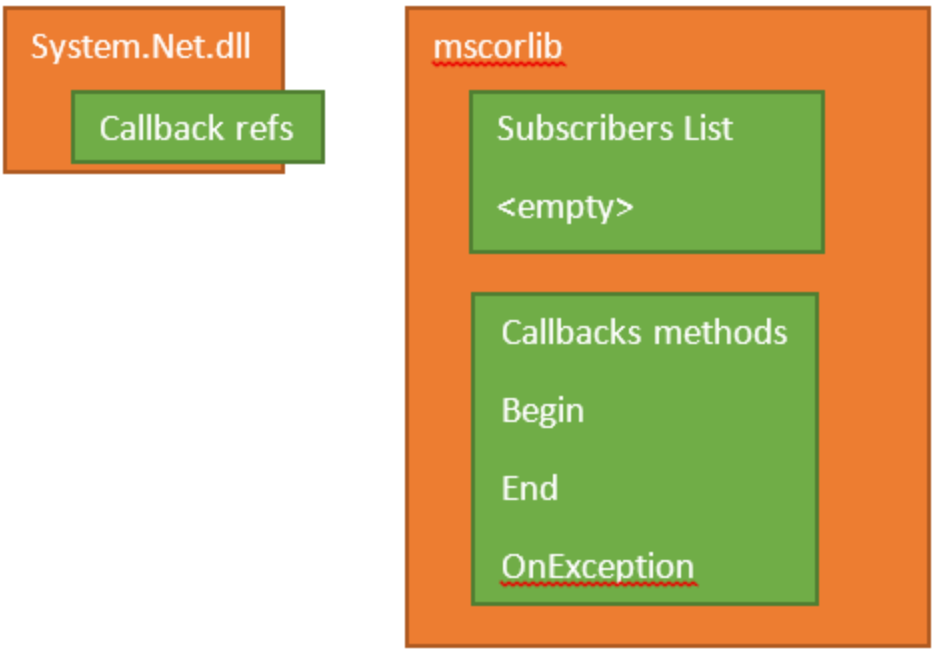
```
    }
    return retVal;
}
```

No GAC references

Intercept will inject assemblies implementing instrumentation callbacks right into system assemblies. Due to the limitation of reJIT technology CLR Instrumentation Engine is pre-loaded with a special extension that will inject code into `mscorlib` This code will implement late binding of the code injected into methods with the callbacks provided by the SDK.

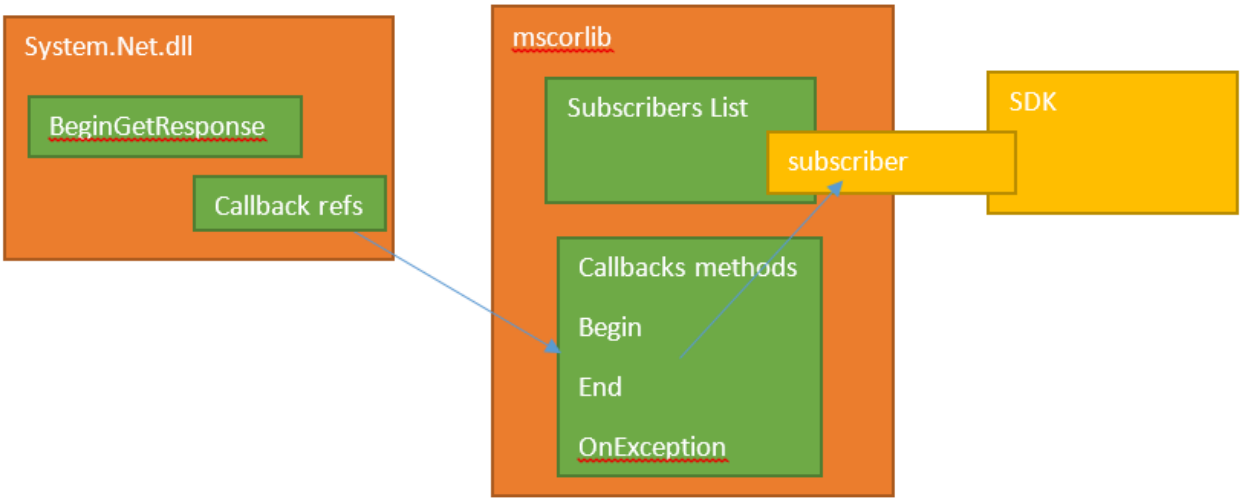
When Instrumentation Engine profiler is not loaded we have `System.Net` and `mscorlib` assemblies:

Once the Instrumentation Engine is loaded it injects callbacks methods as well as a static list of subscribers to these callbacks into `mscorlib`. See [this template]. It also create method references to these callbacks into all modules like `System.Net`.



Once SDK is loaded it requests the Instrumentation Engine to insert callbacks into one of the methods of `System.Net.dll`. This will add provided callbacks into the subscribers list in

`mscorlib` and using method references in `System.Net.dll` will call callback methods from `mscorlib`. These callback methods in turn will call the specified subscriber:



As the same callbacks will be used to instrument different methods (APMC cannot add new callback methods dynamically) additional method identifiers will be associated with subscribers and will be injected into interesting methods (BeginGetResponse). In future CLR may support dynamically adding new methods into already loaded modules. It will help to avoid “marshalling” of the callbacks and lookup in the subscriber’s table.

Multiple AppDomains support

Multiple versions of Intercept may be loaded into the process - one per AppDomain. This allows to version the code injection logic as well as configuration for monitoring based for various apps loaded into a single process via multiple AppDomains.

Loading contexts of .NET Core are not supported.

Summary

CLR Instrumentation Engine is a Microsoft-proposed profilers multiplexing solution. It is and will continue to be pre-installed in many Microsoft environments.

It is already being used for a long time with multiple extensions. However, it is yet to prove it’s maturity with the numerous partner’s profilers simultaneously loaded. Work is ongoing.

Intercept extension enables code injection using the managed code which simplifies development significantly. It is not open sourced, windows-only and does not support .NET Core at the moment. It is NOT currently in active development.

Links

1. https://github.com/microsoft/CLRInstrumentationEngine
2. (Private): https://github.com/microsoft/InstrumentationEngine-Intercept
3. http://apmtips.com/blog/2016/11/18/how-application-insights-status-monitor-not-monitors-dependencies/
4. One interesting application of an Intercept extension: https://www.usenix.org/conference/atc18/presentation/luo