# EventSource Activity Support

**Spec Status:** Complete **Last updated:** 3/13/2019 1:24 PM

| PRODUCT UNIT | | FEATURE TEAM | |
|---|---|---|---|

EventSource is a high speed logging facility for .NET. However in concurrent/asynchronous scenarios typical of server applications, it is critical to be able to 'tag' the events with a correlation ID that allows you to group and filter the events together based on higher level concepts like 'request' or 'session' or 'user'. Historically, the generation and 'flow' of these IDs has been done explicitly by the programmer, leading to complex code, ad-hoc designs, inconsistency, and common errors being repeated again and again. In Version 4.6 of the .Net Runtime, EventSource solves this by doing almost all of the work for the generation and propagation of a general purpose correlation ID, making it trivial to associate events are that are causally connected to high level constructs like 'request' etc.

## Table of Contents

# 1. Introduction: The need for Activity Tracking

System.Diagnostics.Tracing.EventSouce is class that is part of the .NET Framework (Since version V4.5) that is designed as a high performance strongly typed, general purpose logging system. It is also available as a standalone Nuget package, so it can be used on earlier versions of the framework. See this tutorial for more on the basics of using it as well as its specification and users guide. EventSource has a subscription model that allows in-process instances of the EventListener class to subscribe to any events that any EventSource generates (which can then go to files, databases or other logging systems) In addition the .NET framework has built-in support for sending EventSource events to the Windows Event Tracing for Windows (ETW) logging system. This allows tools like PerfView and Windows Performance Analyzer (WPA) to subscribe the events from outside the process and to display the events in useful ways.

While EventSource does a very good job at allowing code to log **individual events** carrying a wide variety of data payloads in a high performance, type safe way, it needs an additional features to support correlating events **among each other** in an easy way. This this commonly referred to 'Activity Tracking' and the extra information in the events that supports it are often called Activity IDs. This is the subject of this specification. However building an easy-to-use correlation system is harder than it seems, and so a bit of background is needed to even frame the problem. This document provides this background as well as the new features in .NET Version 4.6 that does Activity Tracking mostly completely automatically.

## 1.1 Why You need Activity Tracking

Consider the very simple scenario of a server that responds to HTTP requests. Each request goes through several phases

1. A HTTP Request is received
2. A security validation is initiated on the incoming request.
3. The security validation completes and typically is successful.
4. Several database lookups are done during request, each one takes a noticeable amount of time (MSec).
5. The Response is sent, and the Request ends.

It is very natural to want to instrument this code so that you can measure performance and diagnose issues when things go wrong. The obvious way of doing this is to create events that mark significant events. Using EventSource we would probably define a set of events like this

```
[EventSource(Name="MyCompany-MyService")]
public class MyServiceEventSource : EventSource
{
    public static MyServiceEventSource Log = new MyServiceEventSource();

    public void RequestStart(string url) { WriteEvent(1, url); }
    public void RequestStop(bool success) { WriteEvent(2, success); }

    public void SecurityStart(string userName) { WriteEvent(3, userName); }
    public void SecurityStop(bool success, string errorMessage) { WriteEvent(4, success, errorMessage); }

    public void DatabaseCommandStart(string dataBase, string command) { WriteEvent(5, dataBase, command); }
    public void DatabaseCommandStop(bool success, string errorMessage) { WriteEvent(6, success, errorMessage); }
}
```

And we would sprinkle calls to EventSource logging throughout our code like this

```
static void ProcessOneRequest()
{
    // receive a request
    MyServiceEventSource.Log.RequestStart(request.url);
    // ...
    // Validate secruity
    MyServiceEventSource.Log.SecurityStart(security.userName);
    // ...
    MyServiceEventSource.Log.SecurityStop(security.success, security.errorMessage);

    //... Do one or more data base calls
    MyServiceEventSource.Log.DatabaseCallStart(dbComand.dbName, dbCommand.command);
    // Perform the call
    MyServiceEventSource.Log.DatabaseCallStop(dbCommand.success dbCommand.errorMessage);

    // Process the web page, issue the response and then
    MyServiceEventSource.Log.RequestStop(success, errroMessage);
}
```

Now obviously this is pseudo code as real code would embed these calls at the appropriate level of abstraction (inside specialized classes), but we can ignore that for this discussion.

We immediately notice several things

1. It is VERY common that we need to describe something that has a duration. An event however does not do this naturally, it represents a particular INSTANT of time. We call these things with duration activities.
2. While you CAN represent an activity by a single event at Stop event with a 'duration' property, this is pretty inconvenient when processing the trace because you have to 'look ahead' in the trace to find when things start (and you really don't know how far ahead to look). It is much more natural to simply log two events, on at the 'start' and the other at the 'stop' of the activity.
3. Because activities have duration, they can now NEST (something individual events could not do), and in fact they can overlap in complex, non-nesting ways as well. Because of this activities are significantly more complex than events.
4. If your process is truly single-threaded, **and your activities always properly nest**, then you don't need additional correlation information in the events. You can match up a stop with its corresponding start by looking for the last unstopped event for the same activity type (e.g. Request, Security, DatabaseCall). In this way you can unanimously determine at any point of time what activities are active.
5. If, however, you have any kind of parallelism possible in the system (typical on servers), events without additional correlation information become ALMOST USELESS, because you know longer know which stop event is for which of the concurrently active start events.

It is this last observation which is the crux of the problem we are trying to solve. In a server environment, you need a way of matching up start and stop events, as well as a way of determining if the other events and activities nest inside the start and stop events. If each request was GUARENTEED to only execute on a single thread, and that thread ID was captured as part of the event (as it is in ETW), then the THREAD-ID could server this purpose. Effectively you should separate all events by thread first, and then use the normal single-threaded technique to analyze the trace from there. Unfortunately, real code does NOT make this guarantee that all related work happen on a single thread, and with the advent of asynchronous programming, it is PROFOUNDLY untrue. Thus in normal, common server scenarios, events without correlation information are next to useless.

So it comes down to this

**Cross event correlation is indispensable in a server environment.**

# 2. Using EventSource Activity Support

In a later section, we will talk about some of the alternatives to the approach that EventSource took to provide event correlation, however, rather than start out with the 'bad' ideas, we start with presenting the 'good' one, which EventSource adopted.

## 2.1 End User Experience

From an end user experience point of view, to a very good approximation, EventSource activity support is an 'it just works' feature. In particular the EventSource scenario described in section 1 DOES NOT CHANGE AT ALL, and yet, now when processing the event data, you now have acesss to an 'ActivityID' that allows you to correlate events in the way you wish to. For example, if you ran the application on Version 4.6 of the .NET runtime, and used version V1.8 of PerfView or higher to collect and view the EventSource data, you would get a view of events that would look like this.

| Event Name | TimeStamp | ThreadID | ActivityID | DURATION_MSEC |
|---|---|---|---|---|
| MyCompany-MyService/Request/Start | 6,619.23 | 3,576 | //1/1/6/1 | |
| MyCompany-MyService/Security/Start | 9,403.14 | 6,228 | //1/1/6/1/1/2 | |
| MyCompany-MyService/Security/Stop | 9,723.26 | 6,228 | //1/1/6/1/1/2 | 320.112 |
| MyCompany-MyService/DatabaseCommand/Start | 12,214.79 | 4,508 | //1/1/6/1/2 | |
| MyCompany-MyService/DatabaseCommand/Stop | 12,215.13 | 4,508 | //1/1/6/1/2 | 0.341 |
| MyCompany-MyService/DatabaseCommand/Start | 13,085.57 | 8,916 | //1/1/6/1/3/1 | |
| MyCompany-MyService/DatabaseCommand/Stop | 13,085.68 | 8,916 | //1/1/6/1/3/1 | 0.106 |
| MyCompany-MyService/Security/Start | 13,085.79 | 8,916 | //1/1/6/1/3/2 | |
| MyCompany-MyService/Security/Stop | 13,394.61 | 8,916 | //1/1/6/1/3/2 | 308.821 |
| MyCompany-MyService/Request/Stop | 15,385.33 | 8,196 | //1/1/6/1 | 8,766.093 |

The new information in V4.6 is the 'ActivityID' and DURATION_MSEC in the output above. The first thing you notice is that the ActivityID is not a 'flat' number but something that looks much more like a path (e.g //1/1/6/1). This is because each activity has a 'parent' that 'caused' it, and these causal relationships naturally form tree, and the natural ID for a node in the tree is the path from the root of the tree to a node. These causal relationships also correspond to nesting relationships among the activities. Thus we see that the 'Security' activity (with Id //1/1/6/1/1/2) must be 'caused' (part of or nested inside) the 'Request' activity (with Id //1/1/6/1) because //1/1/6/1 is a prefix of //1/1/6/1/1/2.

For example, filtering so that you only see ALL events associated with a particular request (e.g. like //1/1/6/1) is as simple pattern matching on the //1/1/6/1 prefix. In fact the trace that the data above came from had 8 concurrent requests, but we could easily filter to see just one of them by this simple prefix test.

Once all events have such an ID, it is a simple matter to compute durations by subtracting the timestamp for the start from the timestamp for the stop. This is what the DURATION_MSEC column is.

The view above emphasizes the chronological order in the events. It is also possible to display the information that emphasizes the hierarchical nesting of the activities. Shows the PerfView view where events have been organized into a tree view based on their 'activity path' (This is PerfView's Any StartStopTree View)

| Name ? | Inc Ct ? | First ? | Last ? |
|---|---|---|---|
| ROOT | 4 | 12,214.788 | 12,214.950 |
| + Process64 TaskTesting (3804) | 4 | 12,214.788 | 12,214.950 |
| + Activities | 4 | 12,214.788 | 12,214.950 |
| + Activity Loop(//1/1) | 4 | 12,214.788 | 12,214.950 |
| + Activity ForkJoin(//1/1/6) | 1 | 12,214.788 | 12,214.788 |
| + Activity ForkJoin(//1/1/3) | 1 | 12,214.821 | 12,214.821 |
| + Activity ForkJoin(//1/1/7) | 1 | 12,214.847 | 12,214.847 |
| + Activity ForkJoin(//1/1/8) | 1 | 12,214.950 | 12,214.950 |
| + Activity Request(//1/1/8/1) | 1 | 12,214.950 | 12,214.950 |
| + Activity DatabaseCommand(//1/1/8/1/2) | 1 | 12,214.950 | 12,214.950 |
| + Thread (5368) CPU=194ms | 1 | 12,214.950 | 12,214.950 |
| + Event MyCompany-MyService/DatabaseCommand/Start | 1 | 12,214.950 | 12,214.950 |

As can be seen very clearly in the tree view, there was an activity //1/1/ called 'Loop' which started four 'ForkJoin' activities, one of these (//1/1/8) was opened in the view to show that it caused a Request, which in turn caused a database command. Effectively the view above is showing you the 'tree' of activities that is formed by activity ID paths. This allows us to organize and filter data associated with activities in very intuitive ways.

Thus EVERY event (whether it comes from EventSource or not), can now be 'tagged' with this Activity ID and thus be put into this tree. This allows us to group any other metric (CPU time, Disk I/O, Memory Allocation, Clock time, etc) according to these high level activities which is generally what you wish to do.

This becomes VERY powerful, because information on one event (e.g. a user name, or a URL, or a machine name) can be associated with other events BECAUSE THE ARE ON THE SAME (or nested) activity. This is absolutely key to doing sophisticated analysis and is all made possible by the lowly hierarchical activity ID.

## 2.2 Limitations

There are a number of limitations that you need to be aware of when using the feature

1) If your code cause one thread to do work on behalf of another and does NOT use System.Threading.Tasks.Task to do this, then you will not get automatic activity flow through that part of your code. The guidance is not to do this (use Tasks!), however you may have old code that uses old style Async or worse, 'home grown' worker threads. Some of these MAY be handled in the future, but you may need to modify the code to get good results.
2) As we will see we, causality tracking gives up if activities that start the same thread/task do not properly nest (one of the activities is truncated). In particular this can happen if the PARENT of a concurrent task logs the 'start' request. Don't do this. Instead have the CHILD task log the start event (it is actually more natural anyway).
3) As we will see, if any activity is recursive, you must declare these explicitly as recursive activities and use extra care (see section on Recursion).
4) This feature does not address cross process correlation (but see Cross-Machine correlation section).

5) The ActivityID being used by this tracking mechanism is the same ActivityID that is set by using SetCurrentThreadActivityId API. If users use this API correctly (sending a transfer event before changing it and reset the activity ID before returning to unknown code), there should not be interference, however it is possible that 3rd parties have improperly used SetCurrentThreadActivityId and could break EventSources use of the ActivityID field.

### 2.3 In Summary

- To take advantage of EventSource Activity Tracking you need only suffix your events with 'Start' and 'Stop'. No further changes in the code are necessary.
- The result is that every 'Start' operation generates a new ID that is a 'child' of the code that called 'Start' and all subsequent code that logically follows from that code inherits that ID. This code can in turn log other start events which in turn have children, and thus form a tree of activities, each of which has a unique ID that is attached to ANY event that is generated, including the start and stop events themselves.
- Tools that process the events can then use these hierarchical IDs to perform useful grouping and filtering operations (such as viewing all events associate with a particular request).
- Having such an idea is the gateway for tools to do further 'sharing' of information from one event (say the 'username' field of a 'login' event) to another event (say SQL request to a particular database), allowing useful analysis to be done (say a table of database usage broken down by user).

## 3. Details of turning on Activity Tracking

Activity Tracking does have a cost. Once start events fire, the task library has to generate a new GUID and it has to track this GUID across any Tasks that's are created from the current task. It is not a large overhead, but if you generate many tasks per second (e.g. > 10K), it will become noticeable. For this reason you have to 'opt-in' to activity Tracking. The way this is done is with a Keyword on the Task Library EventSource.

The Task library has an EventSource called

Name: System.Threading.Tasks.TplEventSource
Guid: 2e5dba47-a3d2-4d16-8ee0-6671ffdcd7b5

(Tpl stands for Task Parallel Library, and old name for the Task library). Sadly this EventSource predate the convention on naming (otherwise it would have been called Microsoft-DotNet-Tasks) and generating the GUID from the name. As a result, you pretty much have to use the GUID when interacting with this EventSource through ETW.

There is a keywords for this EventSource are

```
public enum Keywords : long
{
    Tasktransfer = 0x1,
    Tasks = 0x2,
    Parallel = 0x4,
    Asynccausalityoperation = 0x8,
    Asynccausalityrelation = 0x10,
```

```
        Asynccausalitysynchronouswork = 0x20,
        Taskstops = 0x40,
        TasksFlowActivityIds = 0x80,
    };
```

And one that is of interest to us now is the TasksFlowActivityIds. When this keyword is enabled, then the task library will flow the activity IDS so that Start-Stop activity tracking works.

Now in the case of PerfView, this provider and keyword are enabled by default, and so it will 'Just Work' However for WPR you will have to enable it explicitly by using the provider GUID 2e5dba47-a3d2-4d16-8ee0-6671ffdcd7b5 and the Keyword 0x80.

The TraceEvent library has a class called TplEtwProviderTraceEventParser which represents the Task library EventSource, so you can turn on this provider/keyword with the following code.

```
      var session = new TraceEventSession("MySession", "MyFile.etl");
      session.EnableProvider(TplEtwProviderTraceEventParser.ProviderGuid,
TraceEventLevel.Informational,
                             TplEtwProviderTraceEventParser.Keywords. TasksFlowActivityIds);
```

# 4. How EventSource Activity Tracking Works

Note that the material in the section is not really needed to use the Activity tracking feature (that was covered in the previous section), however there are corner cases (e.g. recursion) and issues about error handling that can only be properly presented by describing exactly how the activity tracking works. That is what we do here.

The basic design intuition for EventSource Activity tracking was stated before: a thread ID works great as a correlation ID if you could be sure that all things that you could possibly be related to an event happens on the same thread. The insight is that we may be able to get back to this simple case by finding something like a thread ID that DOES work in a concurrent/asynchronous environment.

The basic attribute of a thread that makes it the natural correlation ID is that is that it represents causality. Fundamentally things that happened 'before' in the thread of execution 'causes' the things that happen later. As long as we can 'flow' this causality from one piece of code to any code that is logically caused by that code, we will have the property we need.

For this we rely rather heavily on the System.Threading.Tasks.Task class. A Task can be thought of as an independently dispatchable unit of execution. This is much like what a thread is but much lighter weight. It is best thought of a small snippet of some thread's execution that has well defined boundaries (it begins and ends).

The important property of Tasks is that the .NET Library STRONGLY encourages all work that is concurrent or asynchronous to be described using Tasks. You are STRONGLY advised NOT to create 'worker' threads that do work on behalf of others, but instead simply create a new task any time you need work done. If everyone follows this advice you get a very nice property

All code that is logically 'caused' current execution is either

- On the same thread as the current execution OR

- Is in a Task that was created (transitively) from that thread of execution.

The first case is the sequential case that we liked so much because the thread ID could be used as a correlator. The second case is also straightforward since we can keep track who created Tasks. Thus we have our definition of what it means to 'causally flow' a variable.

With this notion of 'causality flow' in mind, we can attempt to describe EventSource's correlation algorithm:

- We define a new type called 'Activity' which holds everything we need to track activities. The most important property of an Activity is its ID (The path we display to the user). It also has a 'Name' which is the name of the event that started it without the 'Start' suffix. Many activities might have the same name (because the same event generated them), but no two activities have the same ID.
- We define a variable called 'currentActivity' to hold the current activity. However currentActivity is not a global variable, it is a new kind of variable called an AsyncLocal variable that has the following properties.
    - If the thread is not executing a Task, the reference to the currentActivity is associated to the Thread (e.g. like a thread local variable). If the thread is executing a Task, the reference is associated with the Task. Thus there are 10 tasks and 5 non-task user threads, there may be as many as 15 copies of the currentActivity variable.
    - Whenever threads or Tasks creates a child Task, we COPY the contents of the AsyncLocal to the newly created child task. In this way value of currentActivity 'flows' to any task that is 'caused' by it.
- When a 'Start' event occurs and the currentActivity is NULL, we generate the ID as follows:
    - The first node in the activity ID's path is the ID for the current AppDomain. AppDomains are concept similar to processes that allows code to run independently of one another. Often programs have only one AppDomain, but if they create more than one, each gets its own set of static variables, and largely run independently of one another. Most programs only have the one default AppDomain which has an ID of 1, and thus most Activity IDs start with //1.
    - There is an AppDomain wide (static) variable that is used to allocate 'top level' activity IDs. This is used to generate the second number in the Activity ID path. Each time a number is allocated the static is incremented to keep the number unique. This number BY DESIGN is an unsigned 32 bit integer, which means that it CAN roll over (insuring that IDs are finite), but is likely to take at least days of activity generation to do so, so the probability of collision while non-zero is extremely low.
- When a 'Start' event occurs and the currentActivity is non-null (we have an existing activity), we generate the ID in a way very similar for the previous case, but instead of using a static variable to get a number we use a 32 bit integer that is associated with the current Activity. We than take this number and concatenate it to the end of the Activity's ID. Thus if a 'Start' happens on Thread/Task that already had the ID //1/5, then the first such start would be given the ID //1/5/1. (And the next allocations on that same Task will generate the id //1/5/2, //1/5/3 …). The resulting ID is also process-wide unique.
- After the 'Start' allocates a new Activity (with a Name and ID), it updates the currentActivity variable with that value. Thus all code that is 'caused' by this code will inherit this Activity.
- We actually log the Start event. Thus a 'Start' event is always the first event to use a new Activity ID.

## 4.1 Forking Behavior of the CurrentActivity Variable

The fact that AsyncLocal variables might have many instances (each associated with a separate thread or task) which can diverge from one another is subtle and worth describing in detail. Consider the following

1) Thread A is executing the Activity //1/5 (that is the 5th top level Start that happened in the first AppDomain) Thus 'currentThread' pointer associated with Thread A points at Activity //1/5.
2) Thread A spawns Task B which causes Task B's distinct currentActivity variable to point to the one and only Activity //1/5. Task B starts running.
3) Thread A spawns Task C which causes Task C's distinct currentActivity variable to point to the one and only Activity //1/5. Task C starts running
4) Task B happens to log RequestStart() first in time. Thus it uses Activity //1/5 to create a ID, and thus gets the ID //1/5/1 and set ITS COPY currentActivity variable to the new Activity //1/5/1.
5) Task C Happens to log SecurityStart() next in time. Its currentActivity is STILL pointing at the original //5/1 Activity and when it generates an ID since //5/1 has already given out ID //5/1/1 it returns the ID //5/1/2 for the ID. Task C then updates ITS COPY currentActivity variable to the new Activity //1/5/2.

Notice that in this sequence Task B and Task C have effectively DIVERGED. Each has their own distinct Activity and any Activity generation is not independent of each other. The stops events can occur in any order and there is no expectation of nesting between any Activity in Task B and any Activity in Task C.

In contrast, if Thread A had executed RequestStart() and SecurityStart() without spawning tasks it would have looked like this.

1) Thread A is executing the Activity //1/5 (that is the 5th top level Start that happened in the first AppDomain) Thus 'currentThread' pointer associated with Thread A points at Activity //1/5.
2) Thread A logs RequestStart(). Thus it uses Activity //1/5 to create an ID, and thus gets the ID //1/5/1. Thread A's currentActivity variable is set to the new Activity //1/5/1.
3) Thread A logs SecurityStart(). Thus it uses Activity //1/5/1 to create an ID, and thus gets the ID //1/5/1/1. Thread A's currentActivity variable is set to the new Activity //1/5/1/1.

Notice in this case, Request Start has ID //1/5/1 and SecurityStart has ID //1/5/1/1 and thus there is the expectation that SecurityStop will happen before RequestStop (they nest). This difference is important for making all the details work out properly.

## 4.2 Stopping Activities – The Propertly Nested Case

It turns out that stops are a bit tricky because although you usually think of activities as nesting nicely, they don't have to (e.g. you can have RequestStart, SecurityStart, RequestStop, SecurityStop). While we have to handle these cases, but they are more complex (and uncommon), so we will handle them later. For now we consider how Stops are handled for the properly nested case.

In the properly nested case, some Thread or Task will execute a 'Stop' and because we are assuming that they are properly nested, it will be the stop for the current activity. In this case, what we do is very straightforward

- After logging the event (which thus the Stop has and Activity ID of the activity being stopped), we simply restore the 'current' activity to what it was before the 'Start' executed.

Thus a properly nested start-stop pair always restores the state of the system to where it was had it never occurred.

At this point we have the broad outline of how the system works. This is the model that you should keep in your head. The rest of the explanation is for hopefully unusual 'corner cases', which are important when you need to understand what is happening in debugging, but hopefully is rarely needed.

## 4.3 Stopping Activities – Non-Nested Case:

So what does happen when you execute a sequence like

```
Log.RequestStart(…)
Log.SecurityStart(…)
Log.RequestStop(…)          // Request stops before security.
Log.SecurityStop(...)
```

On the same thread? The answer is that EventSource considers this an error. In the sequence above it is clear that the Request could not possibly depend on the final result of the security activity since it completes before it. This is true in general. Typically sequences that interleave like this are because the activities are concurrent and independent of one another. If this is true, their corresponding activities should have been executed in its own task (which would have given its own independent currentActivity tracking), and there would be no conflict. Typically the error above happens when the 'parent' logs the start of two activities AND THEN spawns two concurrent Tasks that log the stop. If instead the code was changes so that the 'child Task' logged both the 'Start' and the 'Stop', all would be well. This is our guidance.

So executing non-nested sequences like the one above on the same thread is considered an error. However there is still the open issue of what the system does to recover from that error. Here is what it does

- If the current activity is NOT the activity being stopped, then it begins a search of all currently active activities (that is, have started and not stopped) in reverse order (starting with the most recently started activity). It is searching for an activity with the same name (thus for RequestStop it is searching for RequestStart).
- As it searches, it kills (stops but does not actually emit a stop event) any activity it encounters. The rationale here is that the most likely error is that a user forgot a stop (most likely because an exception bypassed the call that would have logged the Stop), and that the best solution to fix things is to fix things up as if the stop had occurred. We can't actually emit the stop event because that event has payload arguments that we simply don't know. It is assumed that any analysis tool can perform an similar analysis and insert a 'pseudo-stop' if it desires.
- It continues its search until it runs out of active activities, or it finds a matching Start. If it finds the Start then the current activity is reset to the creator of that activity (as you would expect). If no Stop is found then the activity is set to null (the state it was before any 'Start' was executed).
- None of this effects the actual logging of events. Events are still logged, this only affects what Activity ID is stamped on the events.

Thus in the example above, on the first RequestStop it will 'kill' the SecurityStart activity and then stop the RequestStart activity normally

When it gets to the SecurityStop, it will not find that activity to be alive (which is another error), and that stop will simply be ignored.

The net result is that current activity state recovers to what it should be (both activities stopped), but all events between the RequestStop() and SecurityStop() will NOT look like they are in a the security activity (it will look at that activity has stopped). Thus the difference from what you might expect is shown below in red.

```
                    // Events logged with Activity ID //1/5
RequestStart()      // Starts Activity //1/5/1
                    // Events logged with Activity ID //1/5/1
SecurityStart()     // Starts Activity //1/5/1/1
                    // Events logged with Activity ID //1/5/1/1
RequestStop()       // Events logged with Activity ID //1/5/1, (Security killed right before this)
                    // Events logged with Activity ID //1/5 (since Security was killed)
SecurityStop()      // Events logged with Activity ID //1/5 (since Security was killed)
                    // Events logged with Activity ID //1/5
```

The important take-away here is that the system recovers from this user error (effectively by ignoring the non-nested part of one of the activities).

# 5. Error Handling and Recurision

We have already seen one way a user can cause errors in activity tracking (non-nesting start and stops). However there are two other error conditions as well

1) Stop with no corresponding Start
2) Start with no corresponding Stop.

The first error is not that problematic as and we have already seen what EventSource does in that case. It simply ignores the Stop. The second condition is actually much more problematic, because you don't actually know when the error has happened. Consider a trivial case

```
for(int i = 0; i < 10000000; i++) {
    Log.RequestStart(…);
   // Missing Log.RequestStop
    }
```

The problem is that the activity tracking code will assume that there 10 million nested requests and consume a lot of memory waiting for these activities to stop. Worse this kind of error is pretty likely. While unconditionally forgetting to log a Stop is not likely to be common, missing a Stop on error conditions (because a thrown exception bypasses the code that would log the stop) is reasonably likely. Thus as errors occur, and stops don't happen activities 'leak' building up more and more wasted memory. Because there is nothing that ever 'cleans up' these 'orphan starts' over time the leak grows without bound.

This is clearly an unacceptable situation, and the solution that EventSource picked is to DISALLOW RECURSION BY DEFAULT. Thus in the example when the 'RequestStart' is logged, the activity tracker will notice that there is already a RequestStart alive, and automatically stops that activity (also stopping any child activities since we disallow non-nesting as well). Thus rather than this

```
                        // Events logged with Activity ID //1/5
    RequestStart()      // Starts Activity //1/5/1
    RequestStart()      // Starts Activity //1/5/1/1
    RequestStart()      // Starts Activity //1/5/1/1/1
    RequestStart()      // Starts Activity //1/5/1/1/1/1
```

We get

```
                        // Events logged with Activity ID //1/5
    RequestStart()      // Starts Activity //1/5/1
    RequestStart()      // Starts Activity //1/5/2 (Auto stop //1//5/1)
    RequestStart()      // Starts Activity //1/5/3 (Auto stop //1//5/2)
    RequestStart()      // Starts Activity //1/5/4 (Auto stop //1//5/3)
```

This works in most situations very well. It also means that you don't have to be super-careful putting your 'Stop' events in finally causes. It is OK that you don't log the stops of particular activity on error conditions since the exception can be logged as well (it is a standard event in .NET), and its absence can also be inferred from the activity IDs (since there will be no further events on that ID, include the stop). This robustness is very useful.

Disallowing recursive requests is a very good default because the need for recursion is rare. However when you need it EventSource does let you override the default using an EventActivityOptions flag. If RequestStart had been declared like this

```
[Event(1, ActivityOptions=EventActivityOptions.Recursive)]
public void RequestStart(string url) { WriteEvent(1, url); }
```

Then EventSource's auto-stop behavior is suppressed. As we have seen this is dangerous, however it is not nearly as dangerous if the activity is nested inside a normal non-recursive activity. If that is true, then even if you 'lose' RequestStop calls, eventually you will stop the enclosing activity (or start a new instance of that activity) and because of the nesting requirement, all the inner activities will be stopped. Thus there is a 'backstop' that will ultimately cause lost events to be clean up IF TOP LEVEL ACTIVITIES ARE NOT RECURSIVE.

Thus it is strongly advised that if you use the recursive option on an activity that it NOT be top-level (if necessary you make up an enclosing activity just for this purpose).

# 6. Integration with Cross-Machine Correlation mechanism

You can easily see that the Activity Path mechanism is clearly targeted as an INTRA-PROCESS correlation solution. In particular the Activity IDs that are generated are only guaranteed to be unique with a particular process. So the question naturally arises: What about correlation among events in a system that uses more than one process in its end-to-end processing (which it the NORM for mobile and client-server situations).

The answer is to create a 'two-tiered' solution. Now a two-tiered solution may seem like an unnecessary complexity for activity tracking but it has a number of important advantages

1) It is very natural and useful to break a tracing problem down into the 'inter-machine' part (which ignores any complexity within the process and only shows you activities that interact with other machines), and only display the more detailed tracing for those particular components that turn out

to be of interest in a particular investigation. Having IDs that can 'skip' the detailed activity paths (as a two-tiered system would do) is very useful.

2) It is unclear how many bits the 'universal' the ID that tracks cross-machine activities needs to have to represent it. The typical solution here is to generate a 'big random number' (64 bits or more), so that it is 'globally unique' by shear random chance. This is moderately expensive (compared go process local alternatives), and is only needed for IDs that are actually going to be exchanged across machines.
3) In the same way that Activity Paths were useful in the INTRA-PROCESS case, you want your ids to be paths in the inter-process case. This requires them to be 'bigger than a GUID' and frankly it is better to use a string rather than a GUID for representing them. Strings however, are less efficient for ids that only need to be scoped to a single process.

TODO…

The windows group has a correlation vector spec.

In the two-tiered approach only components on that communicate across processes (typically with network packets), need to participate in the top tier. At this tier

1) All incoming and outgoing requests are marked with the top tier activity ID. Activity ID are strings representing a path of numbers (e.g. //34/23/2323).
2) When a request enters the process, it 'extends' the existing ID by concatenating a /1 suffix (thus //2/32/34 becomes //2/32/34/1)
3) Incoming request take the ID and increment the last number in the ID (thus /2/3/34 becomes /2/3/35).

However this tier DOES requires some mechanism for tracking
TODO NOT DONE

# 7. Comparision to Other CorrelationTechniques

This issue of event correlation is an issue for any logging system. Most systems do nothing or very little to support this kind of correlation. Here we outline these and show how problematic they are. Having the logging system deal with this issue is a non-trivial value that EventSource adds.

TODO: NOT DONE

# 8. The Structure of EventSource Activity IDs

Up until now we have been describing the IDs that EventSource generates as a path like //1/5/1. You might think that this ID is actually a string, but it is not. While EventSource is logically independent of operating system, was designed to 'play well' with Event Tracing for Windows (ETW), and ETW used 128 bit GUIDs for activity IDs. EventSource also adopted this convention early in its design, it would be problematic to change it now.

So in fact EventSource Activity IDs are still 128 bit GUIDS, even the ones generated by the activity tracking feature. So how get the paths like //1/5/1?

The answer of course is that the path is 'stuffed' into the GUID. The observation is that

1) Activity Paths (which represent the nesting of activities) don't tend to be 'too deep' 10 is a deep activity path, and 20 should be very uncommon.
2) Most of the number in an activity path are very small (common 1). This is especially true for asynchronous code. Typically one number, is very large (representing the 'top request loop).

Thus we have a standard encoding that generates GUIDS from activity paths (and of course we can decode the GUID back into its path). It is basically a simple variable length nibble (4 bit) at a time packing of a list of numbers, that can encode lists of integers up to length 24 (but a number more like 10-15 is more likely given that one or more of the numbers may be 'big' numbers).

The exact code to encode and decode this format is given in the Appendix, however the broad properties of the algorithm are

1) It has a size limit (paths over a certain length are 'too big')
2) Even when it hits its size limit, it will always return a unique ID (it just no longer encodes the complete path)
3) It tries to encode what it can of the activity path before 'bailing' and suffixing it with something to make it unique.
4) Given any GUID you can detect with high confidence (1 in 4 billion) that the GUID is following this convention or not (it has a 32 bit checksum).

The full syntax for decoded ID looks like this

//1/43/34/343/342323/2322423/43$349734

That is, a sequence of number (separated by /) and then a $ (only present when there was overflow) and then another number that insures it is unique (with the process).

Thus tools like PerfView can look at the standard GUID activity ID and see if it is an Activity Path ID. If it is it decodes it using the algorithm in the appendix and displays it. This gives the user experience that was shown earlier.

## 8.1 Handling Deeply Nested Activites.

It may seem that given GUIDS can only encode a finite activity path that deeply nested activities simply can't be correlated properly but this is not true. It does take more work on the part of the tool doing the processing, but it is possible to handle arbitrarily nested activities.

The key to handling deeply nested activities is the fact that events not only have an 'ActivityID' associated with them they also have obscure field called the 'RelatedActivityID'. One of the important uses of this field is that it

is set by 'Start' events to be the activity ID that actually invoked the start. For example here is the original example which shows the RelatedActivityID as well as the ActivityID

| Event Name | Time MSec | ActivityID | RelatedActivityID |
|---|---|---|---|
| MyCompany-MyService/Request/Start | 6,619.232 | //1/1/6/1 | //1/1/6 |
| MyCompany-MyService/Security/Start | 9,403.142 | //1/1/6/1/1/2 | //1/1/6/1/1 |
| MyCompany-MyService/Security/Stop | 9,723.255 | //1/1/6/1/1/2 | |
| MyCompany-MyService/DatabaseCommand/Start | 12,214.788 | //1/1/6/1/2 | //1/1/6/1 |
| MyCompany-MyService/DatabaseCommand/Stop | 12,215.129 | //1/1/6/1/2 | |
| MyCompany-MyService/DatabaseCommand/Start | 13,085.573 | //1/1/6/1/3/1 | //1/1/6/1/3 |
| MyCompany-MyService/DatabaseCommand/Stop | 13,085.679 | //1/1/6/1/3/1 | |
| MyCompany-MyService/Security/Start | 13,085.788 | //1/1/6/1/3/2 | //1/1/6/1/3 |
| MyCompany-MyService/Security/Stop | 13,394.610 | //1/1/6/1/3/2 | |
| MyCompany-MyService/Request/Stop | 15,385.325 | //1/1/6/1 | |

The value of RelatedActivityID is not at all suprising, it is the activity ID of whatever called the start. Thus star events tell you both the current activity as well as the creator of the current activity. Thus they form a 'linked list' that can tell you the chain of activities with unlimited depth (assuming your logging goes back far enough to see all the relevant start events).

As mentioned previously when activity paths get too long to store in a GUID, we give up trying to store the path but we PRESERVE UNIQUENESS. Thus these IDs are still find 'handles' for finding the matching 'start' event, and from there a tool can use the RelatedActivityID field to find its start, and thus form the entire chain.

Some tools may choose not to bother with this as you do need to track state based on all previous start events, but the information is there for those tools that wish to do this.

# APPENDIX: Activity Path Encoding

This appendix gives details on the encoding of an activity path (e.g. //1/2/522/23) into a GUID.

Basically the GUID is a 128 bit (16 byte) datatype which we divide up as follows

1) 12 bytes, which are further divided into 24 4 bit nibbles, each representing a hexadecimal digit. The nibbles are order high bits 4 bits of a byte first, then the low bits (thus standard Hex representation of the byte represent the order of how the bits are generated).
2) 4 bytes of checksum (which is simply the sum of the previous bytes added as three little endian 32 bit integers and 32 bit ActivityPath-format-ID which has a (arbitrary but now forever fixed) value of 0x599D99AD).

The checksum allows us to rapidly determine if a particular GUID is following this convention or not. (See the IsActivityPath method below).

The algorithm for encoding the list of numbers in the path are as follows it is a very simple prefix nibble code.

1) At the end of the list, emit a 0 nibble, which marks the end.
2) If the next number in the list 0 < N <= 10 then emit the number into the next as the next nibble

3) If N fits in 12 bits and the current Nibble is the high 4 bits, then emit the value 0xC (consuming the upper nibble), and N in the next tree nibbles.
4) If N fits in 8 bits then emit the value 0xC and store N in the next 2 nibbles.
5) If N fits in 16 bits then emit the value 0xD and store N in the next 4 nibbles
6) If N fits in 24 bits then emit the value 0xF and store N in the next 6 nibbles
7) N is limited to 32 bits (so now it must fit), emit the value 0xF and store N in the next 8 nibbles.

The emitter should not emit partial numbers. If the value of N does not fit, then you must emit the overflow value 0xB instead. The next nibble will be 0xC though 0xF followed by the X that makes the GUID unique (as in steps 4-7 above). The emitter must back up sufficiently so that this number fits. Thus it is an error to emit a code that should be suffixed with something when there is insufficient room for the suffix.

Once you have emitted all the nibbles you wish, compute the checksum (as a 32 bit little endian value) and set it for the last 32 bits of the GUID.

## 8.2 Reading GUIDS as activity paths

In the short term, there may be tools that will not display the ETW activity GUID as an Activity Path. However even in this case you can still do searches on activity paths.

The reason is because the encoding for activity paths is a NIBBLE (4 bit) encoding, that happens to correspond to a hexadecimal digit, and the default string representation as hexadecimal chunks. For example the activity path

//1/1/6/1/3/1

If printed in GUID notation would look like

00326111-0000-0000-1-0000befacf59

And you can see that you small IDs from the path in the early blue digits and the checksum in the last red 8 digits. Because a GUID is defined as 1 four byte (little endian) number, followed by 2 two byte (little endian) numbers, follow by 10 one byte numbers, the order can look a bit odd. Thus the first //1 is actually respensentd by the second to last digit in 00326111 because the first nibble is defined by the high order bits of the first byte, and little endian emits the low order bits of the first integer first. Thus //1//1 is represented by 00326111 and the //1/1/6 as 00326111, etc. It certainly is a bit strange, not too hard. When the numbers involved get bigger than 10 (and thus need a prefix code before them) decoding them gets a bit harder, but still not too bad. The main observation is that given any activity path, it is possible to generate a regular expression that would match the GUID syntax. This may be useful in the short term if you work with tools that don't directly support activity paths.

## 8.3 Code for Determining if a GUID is a Activity Path

It is very easy to compute the checksum and thus determine if a particular GUID is an activity path. Here is the C# code to do it.

```
/// <summary>
/// Returns true if 'guid' follow the EventSouce style activity IDs.
/// </summary>
```

```
public static unsafe bool IsActivityPath(Guid guid)
{
    // We compute a very simple checksum which by adding the first 96 bits as 32 bit numbers.
    uint* uintPtr = (uint*)&guid;
    return (uintPtr[0] + uintPtr[1] + uintPtr[2] + 0x599D99AD == uintPtr[3]);
}
```

## 8.4 Code for Decoding a Activity Path

Here is C# code for decoding an GUID into an ActivtyPath String (e.g. //1/3434/23$23232)

```
/// <summary>
/// The encoding for a list of numbers used to make Activity  Guids.   Basically
/// we operate on nibbles (which are nice because they show up as hex digits).  The
/// list is ended with a end nibble (0) and depending on the nibble value (Below)
/// the value is either encoded into nibble itself or it can spill over into the
/// bytes that follow.
/// </summary>
private enum NumberListCodes : byte
{
    End = 0x0,             // ends the list.   No valid value has this prefix.
    LastImmediateValue = 0xA,
    PrefixCode = 0xB,
    MultiByte1 = 0xC,   // 1 byte follows.  If this Nibble is in the high bits, it the high bits of the number are stored in the low nibble.
                        // commented out because the code does not explicitly reference the names (but they are logically defined).
                        // MultiByte2 = 0xD,   // 2 bytes follow (we don't bother with the nibble optimzation
                        // MultiByte3 = 0xE,   // 3 bytes follow (we don't bother with the nibble optimzation
                        // MultiByte4 = 0xF,   // 4 bytes follow (we don't bother with the nibble optimzation
}

/// <summary>
/// returns a string representation for the activity path.  If the GUID is not an activity path then it returns
/// the normal string representation for a GUID.
/// </summary>
public static unsafe string ActivityPathString(Guid guid)
{
    if (!IsActivityPath(guid))
        return guid.ToString();

    StringBuilder sb = new StringBuilder();
    sb.Append('/'); // Use // to start to make it easy to anchor
    byte* bytePtr = (byte*)&guid;
    byte* endPtr = bytePtr + 12;
    char separator = '/';
    while (bytePtr < endPtr)
    {
        uint nibble = (uint)(*bytePtr >> 4);
        bool secondNibble = false;              // are we reading the second nibble (low order bits) of the byte.
    NextNibble:
        if (nibble == (uint)NumberListCodes.End)
            break;
        if (nibble <= (uint)NumberListCodes.LastImmediateValue)
        {
            sb.Append('/').Append(nibble);
            if (!secondNibble)
            {
                nibble = (uint)(*bytePtr & 0xF);
                secondNibble = true;
                goto NextNibble;
            }
            // We read the second nibble so we move on to the next byte.
            bytePtr++;
            continue;
        }
        else if (nibble == (uint)NumberListCodes.PrefixCode)
        {
            // This are the prefix codes.   If the next nibble is MultiByte, then this is an overflow ID.
            // we we denote with a $ instead of a / separator.

            // Read the next nibble.
            if (!secondNibble)
                nibble = (uint)(*bytePtr & 0xF);
            else
            {
                bytePtr++;
                if (endPtr <= bytePtr)
                    break;
                nibble = (uint)(*bytePtr >> 4);
            }

            if (nibble < (uint)NumberListCodes.MultiByte1)
            {
                // If the nibble is less than MultiByte we have not defined what that means
                // For now we simply give up, and stop parsing.  We could add more cases here...
                return guid.ToString();
            }
            // If we get here we have a overflow ID, which is just like a normal ID but the separator is $
            separator = '$';
            // Fall into the Multi-byte decode case.
        }

        Debug.Assert((uint)NumberListCodes.MultiByte1 <= nibble);
        // At this point we are decoding a multi-byte number, either a normal number or a
        // At this point we are byte oriented, we are fetching the number as a stream of bytes.
        uint numBytes = nibble - (uint)NumberListCodes.MultiByte1;

        uint value = 0;
        if (!secondNibble)
            value = (uint)(*bytePtr & 0xF);
        bytePtr++;       // Adance to the value bytes

        numBytes++;     // Now numBytes is 1-4 and reprsents the number of bytes to read.
        if (endPtr < bytePtr + numBytes)
            break;

        // Compute the number (little endian) (thus backwards).
        for (int i = (int)numBytes - 1; 0 <= i; --i)
            value = (value << 8) + bytePtr[i];

        // Print the value
        sb.Append(separator).Append(value);
```

```
            bytePtr += numBytes;        // Advance past the bytes.
        }

        if (sb.Length == 0)
            sb.Append('/');
        return sb.ToString();
    }
```