



Troubleshooting .NET Applications - Knowing Which Tools to Use and When

Abstract

There are three fundamental classifications of problems encountered with distributed applications deployed to production servers. The source of failures, as well as the nature of production environments, needs to be considered when selecting tools for production level monitoring and troubleshooting. This white paper will address two categories of application errors, and will demonstrate the use of ADPlus for pinpointing the root cause of application crashes, and AVIcode Intercept Studio for pinpointing the root cause of functional errors.

This document shall not be duplicated or used for any purposes other than that for which it is being provided. The information disclosed herein was originated by and is the property of AVIcode Inc. and except for rights expressly granted by written consent, such information shall not be disclosed or disseminated in whole or in part. AVIcode reserves all rights hereto. The word AVIcode and Intercept and the AVIcode and Intercept logos are service marks of AVIcode™ Inc. Throughout this document, other trademarked names may be used. Rather than put a trademark symbol in every occurrence of a trademarked name, we state that we are using the names only in an editorial fashion, and to the benefit of the trademark owner, with no intention of infringement.

©2004 AVIcode. All rights reserved. AVIcode and Intercept Studio are trademarked. All other trademarks are the property of their respective owners. 10/04

Introduction

Typically, three types of application problems occur in production:

1. Application Crashes
2. Functionality Failures
3. Logical issues

Application crashes result from memory leaks (where failures to reclaim discarded memory eventually lead to collapse due to memory exhaustion), thread contentions (where a thread makes unreasonable demands on the CPU by competing for resources) and thread deadlocks (where processes are blocked waiting for requirements that cannot be satisfied). Any one of these issues can be severe enough to cause an unexpected shutdown of services, because either IIS restarts the process or the entire system crashes. These problems are severe and affect your entire user population, leaving them frustrated and your help desk flooded with calls.

Functionality failures occur when a particular part of an application does not behave as it was designed to do (such as when a button does not work), experiences an unexpected situation (which equate to unhandled exceptions in the code) and performance degradations. These failures result in complaints from users who do not get what they expected, and users abandoning their sessions completely because of slow response times.

Logical issues tend to be detected and reported by the end user. These problems are only addressable through understanding the business process and making corrections as needed.

This white paper will speak to the first two matters, application crashes and functionality failures, and will outline the best approach for troubleshooting each of them.

Challenges of Production Monitoring

All developers have all debugged applications, using tools built into the IDE or manual techniques like logging. The development environment makes it easy to test out potential software fixes, and then develop a new solution if they fail.

In the production environment, the immediate goal shifts from fixing the problem to providing a solution as quickly as possible *without* disrupting the activities of the end users. This goal is more difficult to attain because:

- Access to the production environment may be limited. For example, security requirements may limit access to logs that contain sensitive data.
- The same tools used in development are either not available or are impractical to use in production.

- Debugging techniques must be non-intrusive. For example, you can not stop IIS in order to debug a single application thread, or stop an application with breakpoints to evaluate an issue.
- Debugging via code changes is impossible in production. Whereas you can iterate multiple times through code changes and testing in development, this practice is too disruptive for a production server.

Debugging Techniques

As we have already discussed, the tools used for production debugging are different from those used in development. Additionally, each category of application issue requires a different debugging technique.

For application crashes, the Microsoft Debugging SDK (ADPlus.vbs, WinDBG, SOS etc) is the best tool for examining thread activity in production at the time a problem occurs. ADPlus monitors the process, traps the error, and creates a dump file, while WinDbg and SOS.dll allow you to peruse the data within the dump file.

For functionality failures, Microsoft EIF (which requires code instrumentation), AVIcode Intercept Studio (which does not require any code modifications) are the best tools for capturing event data. We address debugging these types of failures later in this document.

To find logical bugs, you may use either the application tracing built into ASP.NET, or flight recorders, which chronicle all http traffic and function calls. These tools are difficult to use in a production environment, because they tend to affect system performance by the amount of information that they collect. And because logical bugs are in the eye of the beholder (who may be either the end-user or the business analyst who defined the business logic to begin with), trying to identify the line between bug and feature is difficult.

Debugging Application Crashes

In this section, we will discuss techniques for taking and analyzing memory dumps to diagnose application crashes. Although our examples will concentrate on ASP.NET, the same techniques apply for diagnosing remoting and web services crashes.

Typically, user complaints about application crashes will occur sporadically, with no obvious reason as to why the server has gone down, and no clear way to reproduce the problem. Application crashes manifest themselves in applications returning "Service Unavailable" errors, or in no application response whatsoever. In cases where IIS restarts the worker process, an error may be present in the Application or System Event log that looks similar to this one:

'C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\aspnet_isapi.dll' reported itself as unhealthy for the following reason: 'Deadlock detected'

Deadlocks in DLLs are a common cause of application crashes. Other reasons include high memory usage from memory leaks (a common complaint with .NET applications and a problem which will cause IIS to restart the application), unexpected process terminations (which can occur when mixing native and managed code), or internal Microsoft issues in IIS.

When trying to debug application crashes, there are two scenarios. The first scenario, or the "good" scenario (which we will examine), is when you have access to the server with the issue. The second scenario, or the "bad" scenario, is one where you have no access to the server with the problem. Good luck plays an important role in diagnosing issues under the second scenario.

There are two steps to diagnosing an application crash, the first of which is information gathering and the second is information analysis.

Step 1: Information Gathering Using ADPlus

ADPlus is console-based Microsoft Visual Basic script. It automates the Microsoft CDB debugger by connecting to one or more processes when they start, and then produces memory dumps and log files that contain debug output when they terminate and IIS attempt to restart the process.

ADPlus is designed to debug application crashes due to unhandled exceptions, memory leaks or deadlocks. It also crash mode that allow gathering first chance exceptions.

ADPlus supports the following types of exceptions:

- Invalid handle
- Illegal instruction
- Integer divide-by-zero
- Floating-point divide-by-zero
- Integer overflow
- Invalid lock sequence
- Access violation
- Stack overflow
- C++ EH exception
- Unknown exception

In order to use ADPlus, you must know that you have an application crash and be able to reproduce it. ADPlus can run with either a **-hang** (to troubleshoot processes or applications that stop responding) switch, or the **-crash** switch (to troubleshoot processes or applications that fail)

To create a dump file, run ADPlus in -crash mode against the Aspnet_wp.exe process, and then reproduce the problem. Your symbol path must also point to available PDB files for the application. PDB files permit correlating addresses to names for unmanaged code, and gathering information about line numbers in source code for managed code. Build PDB files with your application in release mode, and keep

them in your back pocket when you release to production in the event that you should need them for troubleshooting later.

Step 2: Information Analysis

While ADPlus is good for collecting information about a variety of crash events, it is not designed for analyzing the information.

To analyze the memory dump information you may use either basic or advanced tools. The basic tools include Visual Studio.NET 2003 (both a managed and native debugger) and CorDbg (a console-based debugger used to debug managed code.)

Advanced tools for analyzing the information include:

- WinDbg - a native debugger with a graphical user interface, and the tool that we will use for our analysis
- SOS - a debugger extension capable of displaying managed call stacks and object data
- CDB - a console program for debugging user-mode applications and drivers
- NTSD - used to debug user mode processes on a test machine

Debugging Functionality Failures

All application crashes are severe, but functionality failures are different, and tools like ADPlus or CDB cannot catch them. If a developer did not write any try/catch code where the functionality failure is occurring, the issue will be unknown to the support staff unless users call and complain.

When some functionality fails, you must have:

- Guaranteed timely and accurate information about the defect
- Enough information to be able to debug, analyze and reproduce the defect
- Guaranteed notifications to the responsible parties

One technique is using EIF (Enterprise Instrumentation Framework) with your code. This framework enables you to gather enough information, but not without writing additional code to do so.

The best solution is one that 1) does not require code changes and 2) provides detailed information at the time the error occurs. AVIcode Intercept Studio meets both of these criteria.

Capturing Exception Events

Frequently an end-user encounters an error page instead of the expected results. The error does not contain any particularly useful information, and if it did, it would require the user to contact someone responsible for the application to relay the

information. There are no error messages in the event log, the network and server are running error free, so your Network Operations staff is happy without knowing that your end user is very unhappy.

The Intercept exception-monitoring agent (X-Mon) attaches to the worker process w3wp.exe when IIS starts much in the same way that ADPlus does.

Microsoft recommends instrumenting the top-most event handler to write out the call stack for analysis in the event that an exception occurs. This troubleshooting method is limited however, because:

- You do not receive any notification of the error unless the end-user calls
- The information in the call stack is static, and does not contain the business context associated with the business transaction. The stack trace only allows you to see that the job failed and frames from both Microsoft .NET methods and from your own code.

The Intercept Studio agent captures information at runtime, and passes that information to Intercept's own web-based structured event viewer, called SE-Viewer.

Intercept collects information about the complete call stack, highlights the frames directly related to the event, and marks in red the frame where the exception actually occurred.

Additionally, the exception-monitoring agent collects the actual values of functional parameters, local variables and member variables, as well as information about the object state, all without having to write a single line of code instrumentation. This is a critical factor to consider when dealing with a production server, particularly one that you may not have access to.

Finally, Intercept provides details about the source code where the exception occurred, and provides a link that will open Visual Studio .NET to the exact line of code with the issue. Notice that the exception information was caught although there is no try/catch block in the code, nor any other special debugging code.

By installing the X-Mon agent you can collect root cause information in production and bring it over to developers in development.

In a typical instrumentation scenario, you need to balance the amount of information collected with the performance impact of the instrumentation itself. This means that you need to know ahead of time what errors to expect, and what type of information you will need to debug the issue. If you do not instrument extensively enough, you may need to add more code to see the values that you need and then redeploy to production. If you instrument too extensively, you may noticeably slow down the response time of your application. This is particularly an issue when writing library functions that other applications will use. Since is no way to predict exactly how the function may be used, you may waste time writing code to handle an exception that developer may not care about or regards as a business exception rather than a critical exception.

Unlike static instrumentation, Intercept Studio's dynamic monitoring allows you to increase or decrease the amount (or depth) of information collected on the fly, and

without affecting system performance. Intercept Studio is flexible enough to allow you to define the number of array members to collect, values for particular objects, add or remove functions to monitor, and differentiate between critical (unhandled) and non-critical (handled) exceptions.

Capturing Performance Events

There are several challenges associated with troubleshooting performance issues in enterprise applications:

- Today's applications span over multiple components, systems, and even sites
- Performance of applications in production varies depending on resources and environment state
- The performance of each component contributes to the overall performance of the application, and thus to the end-user's experience

Application performance tuning is not as simple as it was when applications were all on one box. Distributed applications are affected not only by the performance of each box, but also by the environment, including the state of the network and components that over which you do not have control. In this section, we will discuss performance monitoring in terms of application performance dependencies, the individual application components that participate in distributed business transactions and what factors affect the performance of each component.

The performance of distributed applications is dependent on:

- Application component code execution time (the code itself), which has no serious dependency on the execution environment and can be optimized during development using profilers.
- Resource request execution time, which varies depending on the environment and resource state, and can be monitored using resource specific profilers. These resource requests to fulfill the business transaction actions include SQL, LDAP, Exchange, Web Services and TCP/IP server requests.

The code execution time is static, and can be optimized in development using profilers. Dependencies such as the number of processors and required RAM can all be tested in development.

You cannot test resource request execution time in development. Clients with BMC storage devices find that pre-deployment performance tests do not predict the performance issues that arise with terabyte storage in production. Companies provide dummy blocks to test your transaction formatting against their external credit card processing, but these are poor predictors of transaction performance.

For example, a customer who tries to access an extremely slow page may ultimately abandon the session. He won't call you about the issue, so how will you find out about the problem and fix it before you lose more customers?

By attaching the Intercept performance-monitoring agent (P-Mon), and opening SE-Viewer, we can see a list of performance events for slow performing ASP.NET pages and web service calls. The list includes each event's essential information: duration, application source, server name and occurrence date. By browsing the list, we can see exactly how long it took a transaction to complete.

Intercept allows you to drill down into events to see more details, such as the slowest nodes in the business transaction. These may be expanded to show all nodes in the transaction, and expanded farther to see the actual parameters in the transaction.

To simplify the browsing through this distributed transaction, Intercept displays a graphical chain that allows you to click on various slow nodes and jump into that part of the transaction. This graphical chain provides enterprise view of your application performance.

Intercept pinpoints and provides resource specific details about heavy calls that are causing your application's performance problems, whether they are related to SQL, Oracle, Web Services or other resource requests. For example, you can copy function parameter values associated with a slow SQL call and plug them into a SQL query analyzer.

Intercept collects all of this information without introducing any detectable noise into the performance of your application.

Summary

Debugging distributed enterprise applications in a production environment is different from debugging in a development environment. Production environments require special tools that will not require access to the production server, and will not interfere with the continued operation of either the application in question or other applications on the server.

In this paper, we have addressed how to troubleshoot two of the three types of application problems that occur on production servers: application crashes and functionality failures.

Creating a memory dump with ADPlus and analyzing the dump with WinDbg and the debugger extensions is the best approach for troubleshooting memory leaks, thread contentions and thread deadlocks.

AVIcode Intercept Studio is the best tool for detecting and troubleshooting functionality failures line critical exceptions and performance degradation. Intercept Studio will not impact your production server, and can be tuned on-the-fly to adjust the amount and depth of information collected.

About AVIcode

AVIcode is a software products company with patented technology and innovative products to detect and report application faults in interconnected systems. AVIcode has developed a product line called Intercept Studio, the most complete suite of application monitoring tools for enterprise application monitoring on the market today.

AVIcode designs products to protect your software investment by simplifying maintenance and troubleshooting, dramatically reducing defect resolution time. Intercept Studio detects software crashes, critical exceptions and performance degradations of production enterprise applications. Intercept Studio immediately collects runtime event details and root cause information, and delivers them to the personnel responsible for the health and management of production applications. Intercept Studio is currently in use at Fortune 100 companies, and is rapidly being accepted as the best way to reduce cost and increase customer satisfaction in the ever-expanding enterprise solution market.

Operating since 1998 and incorporated in 2001, AVIcode is a privately held Maryland corporation headquartered in Baltimore, MD.

"Setting the Standard for Application Monitoring"

443.543.5858

www.avicode.com

info@avicode.com

