CHAPTER 25

DTSDIAG

A few times over the years I've faced dire situations that tempted me to return to my former faith. Times of trial and despair are hard on anyone, and my former faith provided just the right crutch to avoid facing reality and ascribe something that was truly unjust or wrong to some higher purpose. But each time I've faced this down—each time I've withstood the temptation—I've found myself stronger and better able to handle the storms of life than before. Freeing oneself from a mental dependence on errant faith is a lot like giving up an addiction—there are powerful temptations to lapse back into the former habits, but the momentary dulling of the senses that comes from falling off the wagon is never worth the high cost.

—H. W. Kenton

I will close out this book by introducing you to a diagnostic application that you may find useful in your own work. It's based on SQL Server's DTS technology and makes use of the DTS object model. It demonstrates the kind of power an application can wield by bringing together the technologies on which SQL Server is based. If you haven't yet read Chapter 20 on DTS, you might want to before proceeding.

The name of the application is DTSDIAG. Its purpose in life is to collect diagnostic data from SQL Server. It can simultaneously collect Perfmon/Sysmon counters; a SQLDIAG report; the application, system, and security event logs; a Profiler trace; and the output of a blocking detection script (as defined in Microsoft Knowledge Base articles 251004, "INF: How to Monitor SQL Server 7.0 Blocking," and 271509, "INF: How to Monitor SQL Server 2000 Blocking").

DTSDIAG consists of a standalone Visual Basic application, four DTS packages, and some miscellaneous command line tools and scripts that it executes to gather the desired diagnostic data. The VB app allows you to specify the version of SQL Server to connect to, as well as the authentication information to use. Once the collection process has been started by clicking the Start button in the app, you can stop it by clicking the Stop button.

I've often found the need for a tool such as this when diagnosing SQL Server issues. Many times, expecting someone to collect Perfmon, Profiler, and the other types of diagnostics that I typically like to look at when investigating an issue turns out to be too much to ask. Often, the person I'm trying to assist simply can't get all the diagnostic collections going at once. Sometimes they can collect the right diagnostics, but they collect them at the wrong times or at different times. DTSDIAG alleviates this by allowing me to configure which diagnostics I need before sending the tool out to a target machine. I set up the types of data I want to collect in an INI file, then have the DTSDIAG executable and support files copied onto the target machine and executed. The only data supplied at the collection site is the name of the server/instance (and version) to connect to and the supporting authentication information. This makes the diagnostic collection process as foolproof as possible while still allowing it to be configured as necessary.

So, now that you know what the app does, let's have a look at its source code. I've already mentioned that diagnostic collection is started/stopped via the Start/Stop button in the DTSDIAG application. Here's the VB code attached to that button (Listing 25.1).

Listing 25.1

```
Private Sub btStartStop_Click()
If Not bRunning Then
  bRunning = True
  btStartStop.Caption = "Stop"
  ExecutePackage "dtsdiag_template.dts", "dtsdiag.dts",
      App.Path + "\dtsdiag.log"

Else
  btStartStop.Enabled = False
  ExecutePackage "dtsdiag_shutdown_template.dts",
      "dtsdiag_shutdown.dts", App.Path + "\dtsdiag_shutdown.log"
  ExecutePackage "dtsdiag_cleanup_template.dts",
      "dtsdiag_cleanup.dts", App.Path + "\dtsdiag_cleanup.log"
  bRunning = False
```

```
  btStartStop.Caption = "Start"
  btStartStop.Enabled = True
End If
End Sub
```

We use the same button for starting and stopping collection and merely change the button's caption based on what state we're in. When we start collection, we call a subroutine named ExecutePackage in order to run the dtsdiag_template.dts package. ExecutePackage saves dtsdiag_template.dts as dtsdiag.dts (I'll explain why in a moment) and runs it.

When we stop collecting, we run two packages: dtsdiag_shutdown_template.dts and dtsdiag_cleanup_template.dts. As with dtsdiag_template.dts, these packages are saved as new packages without the _template suffix and executed.

Certain diagnostics such as the SQLDIAG report and the system event logs can be collected when DTSDIAG is started up or when it is shut down or at both occasions. Whether and when these diagnostics are collected is specified in the DTSDIAG.INI file. The dtsdiag_shutdown_template.dts package exists to collect diagnostics that have been configured for collection during shutdown. The dtsdiag_cleanup_template.dts package exists to remove the stored procedures and other remnants from the collection process once DTSDIAG is stopped. It also checks for the existence of KILL.EXE, a utility from the Windows NT 4/2000 Resource Kit that can terminate other processes, and attempts to kill instances of osql, the utility DTSDIAG uses to collect much of its diagnostic data.

DTSDIAG's configuration file, DTSDIAG.INI, has a very simple format, as shown in Listing 25.2.

Listing 25.2

```
[DTSDIAG]
SQLDiag=1
SQLDiagStartup=0
SQLDiagShutdown=1
EventLogs=1
EventLogsStartup=0
EventLogsShutdown=1
Profiler=1
ProfilerEvents=76,75,92,94,93,95,16,22,21,33,67,55,79,80,61,69,25,
59,60,27,58,14,15,81,17,10,11,35,36,37,19,50,12,13
```

```
Perfmon=1
BlockingScript=1
BlockerLatch=0
BlockerFast=1
MaxTraceFileSize=100
MaxPerfmonLogSize=256
PerfmonPollingInterval=5
ProfilerPollingInterval=5
BlockingPollingInterval=120
Counter0=\MSSQL$%s:Buffer Manager\Buffer cache hit ratio
Counter1=\MSSQL$%s:Buffer Manager\Buffer cache hit ratio base
Counter2=\MSSQL$%s:Buffer Manager\Page lookups/sec
...
```

The format of the file should be pretty self-explanatory. Each type of diagnostic has its own Boolean switch. For example, if the Profiler value is set to 1, we attempt to collect a Profiler trace; otherwise, we don't.

Some of the settings in the file serve as options for the collection process. For example, ProfilerEvents contains a comma-delimited list of events (see sp_trace_setevent in Books Online for the master event number list) to capture in the Profiler trace. The Counter*N* entries contain the list of Perfmon/Sysmon counters to collect. The BlockerLatch and BlockerFast options contain parameter switches for the blocking detection script (again, as outlined in Knowledge Base articles 251004 and 271509).

The key routine in DTSDIAG.EXE is the ExecutePackage method. Let's look at the code (Listing 25.3), then I'll walk you through what it does and how it does it.

Listing 25.3

```
Private Sub ExecutePackage(SrcName As String, TargName As String,
    LogName As String)
  Dim oPkg                As DTS.Package
  Dim oTask               As DTS.Task
  Dim oCreateProcessTask  As DTS.CreateProcessTask
  Set oPkg = New DTS.Package
  oPkg.LoadFromStorageFile SrcName, ""
  oPkg.LogFileName = LogName

  For Each oTask In oPkg.Tasks
```

```
    If 0 <> InStr(1, oTask.Name, "CreateProcess",
        vbTextCompare) Then
      Set oCreateProcessTask = oTask.CustomTask
      oCreateProcessTask.ProcessCommandLine =
          TranslateVars(oCreateProcessTask.ProcessCommandLine)
    End If
  Next

  Dim oFs
  Set oFs = CreateObject("Scripting.FileSystemObject")

  If oFs.FileExists(TargName) Then
    Kill TargName  'Delete in advance so the file won't grow
                   'ad infinitum
  End If

  Set oFs = Nothing

  oPkg.SaveToStorageFile TargName
  oPkg.Execute
  oPkg.UnInitialize
  Set oPkg = Nothing
End Sub
```

The routine begins by instantiating a DTS Package object. Although Package2 is the newer interface (introduced with SQL Server 2000), coding to the Package interface allows us to run on SQL Server 7.0.

Once the Package object is created, we load the specified package from its structured storage file. Each of the packages DTSDIAG uses is stored in COM's Structured Storage File format.

We next iterate through the tasks defined in the package and locate each Execute Process task by searching for CreateProcess in the task's name. We access each Execute Process task by assigning the CustomTask property of the generic task object in the Package.Tasks collection to the previously dimmed DTS.CreateProcessTask variable.

In case you're wondering, we iterate through the Execute Process tasks in each package in order to translate certain placeholders in the ProcessCommandLine property before executing the package. Because we need to execute complex scripts and retrieve their variable output in order to collect diagnostic data via DTSDIAG, we can't use a typical Execute SQL task to run

much of the T-SQL DTSDIAG runs. Instead, we must shell to OSQL.EXE. Obviously, we want our calls to osql to be configurable—for example, we want to be able to specify the server and instance to connect to, the options for some of the diagnostic stored procedures we run, and so on. We could have used one of the custom task objects we built earlier in the book to make this a snap, but that would have required the custom task to be installed on the target machine when packages that contained it were executed. Because I didn't want to require COM objects to be registered before diagnostics could be collected, DTSDIAG doesn't use any custom tasks. Instead, it uses regular Execute Process tasks and placeholders in the ProcessCommandLine property in a manner similar to the ExecuteSQLScript and ExecuteScript custom tasks we built earlier in the book. Our VB code iterates through these tasks and replaces the placeholders with their appropriate values prior to executing a package.

Note the call to the TranslateVars function. TranslateVars is responsible for translating the variables in each ProcessCommandLine into their appropriate values. It's actually more complex than ExecutePackage, and we'll tour it in just a moment.

Once the ProcessCommandLine property for each Execute Process task has been properly translated, we write the translated package to the target package name and execute it. When the package finishes executing, we clean up the package object and return.

As I mentioned, the TranslateVars routine translates the placeholders in each Execute Process task's ProcessCommandLine property into their appropriate values. This means that, for example, it translates %server_instance% into the server and instance to which we want to connect. Similarly, it translates %auth_string% into the appropriate authentication string to be passed on the osql command line.

Some of the values we need to translate come from the DTSDIAG.INI configuration file. Therefore, our code contains a Declare Function DLL import for the GetPrivateProfileString API function, which is the Win32 function used to retrieve values from an INI file. Listing 25.4 shows the source code for TranslateVars and the GetPrivateProfileString import.

Listing 25.4

```
Private Declare Function GetPrivateProfileString Lib "KERNEL32" _
      Alias "GetPrivateProfileStringA" (ByVal AppName As String, _
      ByVal KeyName As String, ByVal keydefault As String, _
      ByVal ReturnString As String, ByVal NumBytes As Long, _
      ByVal FileName As String) As Long
```

```
Private Function TranslateVars(CmdLine As String) As String

  Dim strServer As String
  Dim strInstance As String
  Dim strProfilerParms As String
  Dim strBlockerParms As String
  Dim iBlockerPollingIntervalSeconds As Integer
  Dim iBlockerPollingIntervalMinutes As Integer
  Dim strWork As String

  ' Defaults for INI values
  strProfilerParms = ""
  strBlockerParms = ""
  iBlockerPollingIntervalSeconds = 0
  iBlockerPollingIntervalMinutes = 0

  Const BUFFSIZE = 1024

  strWork = Space(BUFFSIZE)

  ' Get Profiler Parms

  ' Events
  Res = GetPrivateProfileString("DTSDIAG", "ProfilerEvents", "", _
        strWork, BUFFSIZE, App.Path + "\dtsdiag.ini")

  If (0 <> Res) Then
    strProfilerParms = ", @Events=" + Chr(39) + Mid(strWork, 1, _
        Res) + Chr(39)
  End If


  ' MaxTraceFileSize
  strWork = Space(BUFFSIZE)
  Res = GetPrivateProfileString("DTSDIAG", "MaxTraceFileSize", "", _
        strWork, BUFFSIZE, App.Path + "\dtsdiag.ini")

  If (0 <> Res) Then
    strProfilerParms = strProfilerParms + ", @MaxFileSize=" + _
        Mid(strWork, 1, Res)
  End If

  ' Get Blocker Parms
```

```
' BlockerLatch
strWork = Space(BUFFSIZE)
Res = GetPrivateProfileString("DTSDIAG", "BlockerLatch", "", _
      strWork, BUFFSIZE, App.Path + "\dtsdiag.ini")

If (0 <> Res) Then
  strBlockerParms = "@latch=" + Mid(strWork, 1, Res)
End If

' BlockerFast
strWork = Space(BUFFSIZE)
Res = GetPrivateProfileString("DTSDIAG", "BlockerFast", "", _
      strWork, BUFFSIZE, App.Path + "\dtsdiag.ini")

If (0 <> Res) Then
  strBlockerParms = strBlockerParms + ", @fast=" + _
      Mid(strWork, 1, Res)
End If

' BlockingPollingInterval
strWork = Space(BUFFSIZE)
Res = GetPrivateProfileString("DTSDIAG", _
      "BlockingPollingInterval", "", _
      strWork, BUFFSIZE, App.Path + "\dtsdiag.ini")

If (0 <> Res) Then
  iBlockerPollingIntervalSeconds = Val(Mid(strWork, 1, Res))

  ' Since we are plugging the time part, max is 59
  If iBlockerPollingIntervalSeconds > 59 Then
    iBlockerPollingIntervalMinutes = _
        iBlockerPollingIntervalSeconds / 60
    iBlockerPollingIntervalSeconds =  _
        iBlockerPollingIntervalSeconds Mod 60
  End If
End If

' Extract server and instance from ServerInstance TextBox
Dim iPos As Integer

iPos = InStr(1, tbServerInstance.Text, "\")

If 0 <> iPos Then
  strServer = Mid(tbServerInstance.Text, 1, iPos - 1)
  strInstance = Mid(tbServerInstance.Text, iPos + 1)
```

```
  Else
    strServer = tbServerInstance.Text
    strInstance = ""
  End If

  ' Replace tokens
  CmdLine = Replace(CmdLine, "%auth_string%", strAuth)
  CmdLine = Replace(CmdLine, "%ver%", strVer)
  CmdLine = Replace(CmdLine, "%server_instance%", _
      tbServerInstance.Text)
  If taVersion.SelectedItem.Index = 1 Then
    CmdLine = Replace(CmdLine, "%trace_output%", App.Path & _
        "\output\" & "sp_trace.trc")
  Else    'Omit file extension for 80
    CmdLine = Replace(CmdLine, "%trace_output%", App.Path & _
        "\output\" & "sp_trace")
  End If
  CmdLine = Replace(CmdLine, "%server%", strServer)
  CmdLine = Replace(CmdLine, "%instance%", strInstance)
  CmdLine = Replace(CmdLine, "%profilerparms%", strProfilerParms)
  CmdLine = Replace(CmdLine, "%blockerparms%", strBlockerParms)
  CmdLine = Replace(CmdLine, "%bis%", _
      Str(iBlockerPollingIntervalSeconds))
  CmdLine = Replace(CmdLine, "%bim%", _
      Str(iBlockerPollingIntervalMinutes))

  TranslateVars = CmdLine
End Function
```

Once all the required configuration values are retrieved from DTSDIAG.INI, TranslateVars uses the VB Replace function to translate each token into its appropriate value. It finishes by returning the translated process command line as its function result.

You may be wondering why we don't just use a Dynamic Properties task inside the relevant DTS packages since these INI values ultimately end up inside packages. After all, a Dynamic Properties task can retrieve values directly from an INI file without requiring any type of Automation code. Rather than coding an external app that modifies packages on the fly using COM Automation, wouldn't it be simpler just to use a Dynamic Properties task inside each package where we need to read INI configuration values? The answer is that we do use one when possible. However, many of the configuration values we need to supply must be inserted into the middle of task

property values, so they can't readily be supplied by a Dynamic Properties task. Using a Dynamic Properties task to assign an INI configuration value to a property is tenable only when you are assigning the entire property. Assigning only a portion of the property requires a script or external Automation code.

So, now that we've toured the VB source code for DTSDIAG, let's talk about the DTS packages it uses. Open dtsdiag_template.dts (in the CH25\dtsdiag subfolder on the CD accompanying this book) in the DTS Designer so that we can discuss a few of its high points.

The package begins by creating a folder under the startup folder named OUTPUT. If the folder already exists, it is deleted and recreated. This folder will contain all the files collected by DTSDIAG. Output files from tasks we execute to get set up for the collection process (e.g., creating stored procedures) will have ## prefixed to their names. This allows them to be easily distinguished from the actual diagnostic files we're interested in. Normally you can delete these ## files after the collection process is complete. You'll need them only if there is some problem with DTSDIAG.

Note the use of a Dynamic Properties task to load configuration values from DTSDIAG.INI. As I mentioned earlier, we load as many configuration values as we can using a Dynamic Properties task. Each type of diagnostic has a global variable associated with it that controls whether it gets collected. For example, the global variable sqldiag controls whether SQLDIAG.EXE is executed. The Dynamic Properties task sets the sqldiag global variable by reading DTSDIAG.INI and retrieving the value of the SQLDiag key.

The Blocker, Profiler, and SQLDIAG processes within the package begin by calling osql to create the stored procedures they will call to collect the required data. The blocker process creates two stored procedures: one named sp_code_runner, a stored procedure capable of running other procedures or T-SQL code on a schedule or until a logical condition becomes true, and one named either sp_blocker_pss70 or sp_blocker_pss80 (depending on the version of SQL Server you're connecting to), the blocking detection stored procedures provided in the Knowledge Base articles I mentioned earlier. Because the sp_blockerXXXX procedures belong to Microsoft, I have not included them on the CD accompanying this book. You will have to access the aforementioned Knowledge Base articles at http://www.microsoft.com and download them yourself if you want to use DTSDIAG to run them. Alternatively, you can supply your own blocking detection procedure(s)—there's nothing requiring the use of the Microsoft stored procedures in DTSDIAG.

Note that we don't execute SQLDIAG.EXE directly from our DTS package. SQLDIAG.EXE must be run on its host SQL Server; running it

directly from the package would require that the package be run on the server, something you might not want to do. Instead, we call a stored procedure that shells to SQLDIAG.EXE on the server via xp_cmdshell. This allows you to collect a SQLDIAG report without physically being on the SQL Server machine. Note that this technique requires additional steps on a SQL Server 2000 cluster (see Knowledge Base article 233332).

The Perfmon task executes a custom utility I've written in C++ (also included on the CD) that collects a specified set of Perfmon counters and writes them to a Perfmon BLG-format log. PMC is similar to the LogMan utility included with Windows XP and later (see Knowledge Base article 303133) but works on Windows 9x and later as well. Note that, because of a header file change Microsoft made with the introduction of Windows XP, you will need the version of PDH.DLL (the Performance Data Helper library, the engine behind Perfmon/Sysmon) that ships with Windows 2000 in order to use PMC on Windows XP or later. For your convenience, I've included this file in the dtsdiag folder on the CD accompanying this book. If you decide to run PMC on Windows XP or later (as opposed to running LogMan), I recommend that you use the version of the PDH.DLL I've included with DTSDIAG. You shouldn't replace the version of PDH.DLL that comes with the operating system with the one I've included. Just leave it in the DTSDIAG startup folder, and PMC will find it when it starts.

PMC reads the INI file name passed into it as a parameter (DTSDIAG.INI, in this case), locates INI values named Counter*N*, and adds each one to a Perfmon BLG log. If it finds the string "%s" in a counter name, it translates this to the name of the specified SQL Server instance (optionally passed on its command line) before adding it to the Perfmon log. If no instance name is specified, but PMC encounters "%s" in a counter name, it assumes the default SQL Server instance is being specified and replaces the entire "MSSQL$%s" string with "SQLServer" in order to add the counter for the default instance.

The event logs are collected using the elogdmp.exe utility included with the Windows 2000 Resource Kit. Again, since this utility belongs to Microsoft, I haven't included it on the CD accompanying this book. You can actually use any event log dumper utility you want (e.g., dumpel.exe from the Windows NT 4 Resource Kit will also work)—you just need to configure the event log Execute Process tasks accordingly.

Note that the event logs are collected via an Execute Package task, which starts a separate package that collects all three of them in parallel. This is done because event logs are one of those tasks that can be collected at startup or shutdown or both. So, in order to allow for event log collection from dtsdiag_template.dts as well as dtsdiag_shutdown_template.dts, we've

put the event log collection tasks off in their own package, which we execute as appropriate during startup or shutdown.

A final aspect of DTSDIAG that's worth exploring is the way we use ActiveX script workflow associations to enable/disable certain execution paths within packages. You'll recall that we discussed this technique earlier in the book. In DTSDIAG we use it, for example, to disable the Profiler task path when the DTSDIAG.INI Profiler value is set to 0 (false). Listing 25.5 presents the ActiveX script that's associated with the Create Profiler Proc Execute Process task.

Listing 25.5

```
Function Main()
  If DTSGlobalVariables("profiler") Then
    Main = DTSStepScriptResult_ExecuteTask
  Else
    Main = DTSStepScriptResult_DontExecuteTask
  End If
End Function
```

The global variable profiler is assigned by the Dynamic Properties task at the start of package processing. If this variable is nonzero, we execute the Profiler task path, otherwise, we skip it.

For tasks that can be executed at startup, shutdown, or both, we have to check a second global variable to determine whether to execute them. Listing 25.6 shows the ActiveX script associated with the SQLDIAG task line.

Listing 25.6

```
Function Main()
  If (DTSGlobalVariables("sqldiag")) And _
  (DTSGlobalVariables("sqldiagstartup")) Then
    Main = DTSStepScriptResult_ExecuteTask
  Else
    Main = DTSStepScriptResult_DontExecuteTask
  End If
End Function
```

So, we check not only the global variable sqldiag but also sqldiagstartup (or sqldiagshutdown) to be sure that we're supposed to collect the SQLDIAG report when this particular step is executed. In the dtsdiag_template.dts, we check sqldiagstartup; in dtsdiag_shutdown_template.dts, we check sqldiagshutdown.

That's DTSDIAG in a nutshell. You can run the utility to experiment with it further and load its various packages into the DTS Designer to see how they're constructed. You can play with the VB code to explore controlling DTS packages via Automation. The source code and support files for DTSDIAG are located in the CH25\dtsdiag subfolder on the CD accompanying this book.

A natural evolution to the DTSDIAG concept is the notion of loading the collected data into SQL Server for analysis. I will leave that as a reader exercise but will provide a few hints for the adventurous. The event log and SQLDIAG reports are plain text files and, with some massaging, can be easily imported into SQL Server tables. The blocking script output can also be processed as text and imported into a set of SQL Server tables, although it's a little more challenging because of the variability in the output format. A Profiler trace can be read as a rowset using the fn_trace_gettable T-SQL function, so importing it into a table is a snap. A Perfmon BLG log can be converted to a CSV format using the Relog tool included with Windows XP and later (see Knowledge Base article 303133), which can then be imported into SQL Server using DTS. Once you have all the data in a SQL Server database, you can dream up all sorts of sophisticated analysis for it. The trick is in coalescing the data in the first place.