Real World SQL Server Administration with Perl

LINCHI SHEA

Apress™

Real World SQL Server Administration with Perl

ISBN (pbk): 1-59059-097-X

Printed and bound in the United States of America 12345678910

Technical Reviewer: Mark Allison

Editorial Board: Dan Appleman, Craig Berry, Gary Cornell, Tony Davis, Steven Rycroft, Julian Skinner, Martin Streicher, Jim Sumser, Karen Watterson, Gavin Wright, John Zukowski

Assistant Publisher: Grace Wong

Project Manager: Tracy Brown Collins

Copy Editor: Kim Wimpsett

Production Manager: Kari Brooks

Production Editor: Janet Vail

Compositor: Argosy Publishing

Indexer: Carol Burbo

Artist and Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 11

Monitoring SQL Servers

ONE OF THE CARDINAL RULES every SQL Server Database Administrator (DBA) will inevitably learn is "Never let your users tell you there's a SQL Server problem." If you want to run a smooth SQL Server operation and always want to be able to tell your users that you have the situation under control when they call, you need to monitor your SQL Servers closely and effectively.

This chapter discusses the Perl solutions to some common SQL Server monitoring problems and presents a suite of Perl scripts that together form a comprehensive toolset for detecting SQL Server problems and alerting you of them.

Let's pause to consider the nature of SQL Server monitoring. Broadly speaking, you need two types of monitoring in place. One is proactive monitoring, which is to identify system states or behavior patterns that could result in the disrupted database service, unhappy users, or even lost business if you don't take any corrective action.

Proactive monitoring may include, for instance, monitoring disk space usage so that you don't run out of space unexpectedly when the database is being used during the business hours and so that you can effectively conduct disk capacity planning. Proactive monitoring may also include checking the SQL Server configurations for best-practice compliance. In general, any aspect of SQL Server that has a potential to degrade the quality of the database service you provide to your user community should be monitored proactively.

The other type of monitoring is exception monitoring, which means to alert the DBA of any critical event or error condition—in other words, an *exception*—that has already taken place in the SQL Server environment. The following are the most common SQL Server exceptions:

- A significant message is recorded in the SQL Server errorlog.
- The server becomes unreachable.
- The SQL Server instance becomes unavailable.
- A database becomes unusable.
- The SQL Server Agent becomes unresponsive.
- The SQL Server cluster experiences a significant state change.

Other events or error conditions that you could consider exceptions include an excessively large errorlog, a failed SQL Server Agent job, a failed replication agent, and unusable SQL Mail.

The focus of this chapter is on monitoring SQL Server exceptions, particularly the ones identified in the previous list.

Proactive monitoring, although extremely important, is too broad a topic for a single chapter because you should proactively monitor every aspect of SQL Server. However, as you read this book, you'll notice that proactive monitoring is in fact promoted throughout the entire book.

It should be noted that the exceptions identified in the previous list are the exceptions to the normal SQL Server database service you provide to the users, which ultimately is to allow them to submit queries and receive results within an acceptable response time. An exception may be caused by a variety of root problems such as hardware failure, excessive resource consumption, fatal software bugs, and so on. When it comes to SQL Server monitoring, you're not so much concerned with immediately identifying the root cause of an exception as getting notified promptly and accurately. Only after you've been notified of an exception can you then start to troubleshoot its root cause.

Using Perl Scripts vs. Third-Party Tools

Numerous third-party monitoring tools are on the market ranging from ones that work exclusively with the Windows event logs to comprehensive enterprise-scale packages that support plug-in modules for monitoring various systems or applications including SQL Server. Until you've tested them carefully or have tried them in production environments, the sales brochures may lead you to believe that any of them could satisfy your entire SQL Server monitoring needs.

Don't believe that! Your local requirements will sooner or later outgrow the limited configuration options in any third-party tool, making it necessary to request additional customization either from the vendor or the in-house specialists of that tool. Such requests are often painful and time consuming to get fulfilled. It's also common that a tool may be right for your requirements, but it turns out to be much too expensive for your organization.

Fortunately, writing robust Perl scripts to monitor SQL Server and notify the DBA of exceptions isn't a monumental task and is in fact quite doable even for a busy DBA. When designed properly, the scripts can be highly configurable and can be conveniently modified to accommodate almost any monitoring requirements you may fancy. You can write a suite of monitoring scripts at a fraction of the cost for purchasing an equivalent third-party solution.

I should note that this is all dependent on your comfort level with Perl and the attitude of your management toward a tool such as Perl or toward scripting in general. However, even in an environment hostile toward Perl and friendly toward purchasing off-the-shelf tools, Perl scripts for monitoring servers can be effectively used to complement the officially sponsored monitoring solution, filling the inevitable cracks that will develop between these tools and your requirements.

This chapter focuses on exception detection—finding what exceptions have occurred and making decisions on notification. It won't dwell on the actual delivery of notifications. For exception monitoring, the most common way to deliver an urgent notification is to send a message to a pager or a similar device. Different environments may have different automated paging infrastructures. It can be the low-level socket interface to a paging server or a higher-level email-to-paging gateway. You may also find command-line utilities for sending messages to a pager. For this chapter, all notifications will be delivered via Simple Mail Transfer Protocol (SMTP) emails to their intended pagers. Sending email messages to a pager has worked well in practice.

NOTE *Beware of the inherent delay in an email system. If you find that too many of your alerts take too long to reach their destinations, you should consider an alternative mechanism to deliver your alerts.*

This chapter builds three major monitoring tools in multiple steps. These three tools are for monitoring SQL Server errorlogs, SQL Server availability, and SQL Server cluster status.

First, the chapter starts by building a bare-bones script for monitoring errors recorded in a SQL Server errorlog. Then, it adds more features to the script to improve its robustness and usability and to make it a comprehensive tool for monitoring errorlogs in an enterprise.

Then, the chapter presents a basic script to check whether a server is available and whether the script can connect to the SQL Server instance. The chapter adds more availability checks including querying the databases and checking the SQL Server support services, such as SQL Server Agents, as well as improving the usability of the script.

Finally, the third tool monitors whether a SQL Server cluster has experienced a significant state change. Again, this chapter develops a basic version first and then a more robust and comprehensive version.

These three scripts lay the foundation for a comprehensive SQL Server monitoring infrastructure. This chapter uses a simple script for monitoring critically low free drive space to illustrate how other monitoring tasks can take advantage of this infrastructure.

Monitoring SQL Server Errorlogs: The Basic Version

> **SCENARIO**
> *You want to be alerted when your SQL Server errorlog records a critical error.*

For now, let's assume a critical SQL Server error in the errorlog is an error with severity level 17 or higher. In a SQL Server 7.0 or 2000 errorlog, the message text of a SQL Server error normally adheres to a common format similar to this one:

```
2002-07-12 23:39:59.58 spid51 Error: 15430, Severity: 19, State: 1 ...
```

You can therefore write a script to scan the errorlog at regular intervals for any new entries matching this format. If there's a match and the severity level is 17 or higher, you've found a critical error. Then, you need to decide whether to send a notification.

You obviously don't want to send a notification for every error of severity level 17 or higher in the errorlog. This will flood the pager when errors are generated repeatedly at a high frequency on the SQL Server instance, and it's not at all uncommon to see the errorlog filled with the entries of the same error (in other words, entries with the same error number and severity level). For instance, when the database space is used up and can't be automatically expanded, in the errorlog of a busy server you may see a barrage of SQL Server error 1105 or 9002, complaining about data files or log files being full.

On the other hand, you don't want to insist on sending the next alert only when it's a different error. If the same error is still being logged to the errorlog half an hour or one hour after the first notification is sent, either the DBA hasn't received or responded to the first alert or the problem is still being worked on and hasn't yet been solved. Because there's no way to automatically tell which is the case from a script, another notification of the same error is in order.

You can avoid flooding the DBA with repetitive alerts on the same error with a combination of the following mechanisms:

- Only recording the last entry for the same error during a scan
- Controlling the frequency of scanning the errorlog
- Controlling the frequency of alerting on the same error after it has been detected

The issue of how you implement these mechanisms will become clear momentarily when you see the script `alertErrorlog.pl` (in Listing 11-1). First, let's use an example to see how you can run this bare-bones script to monitor a SQL Server errorlog for critical errors.

Monitoring an Errorlog

To monitor the errorlog of SQL Server instance SQL1\APOLLO, you can schedule to run the script `alertErrorlog.pl` as follows (with the command and all the parameters on a single line) at a regular interval:

```
cmd>perl alertErrorlog.pl  -e \\SQL1\e$\mssql\mssql$apollo\log\errorlog
                           -a sql@linchi.com
                           -i 20
                           -s d:\dba\log\status.log
                           -S SQL1
                           -r 4321557@myTel.com
                           -m mail.linchi.com
                           -q 22-7
```

The script `alertErrorlog.pl` accepts the command-line arguments described in Table 11-1.

Table 11-1. The Command-Line Arguments Accepted by `alertErrorlog.pl`

COMMAND-LINE ARGUMENT	DESCRIPTION
`-a <SMTP account>`	Specifies the account of the SMTP email sender.
`-e <Errorlog>`	Specifies the SQL Server errorlog in Uniform Naming Convention (UNC).
`-i <Min alert interval>`	Specifies the minimum amount of time in minutes between two consecutive alerts for the same error.
`-s <Status log file>`	Specifies the file to which the data structure representing the errorlog monitoring status will be saved. The saved information gives the script memory of what has taken place and helps it make better notification decisions.
`-S <SQL Server name>`	Specifies the name of the SQL Server instance to be identified in the notification message.
`-r <Recipient>`	The email address of the alert recipient. Often, this is the email address of a pager.
`-m <SMTP server>`	The SMTP mail server, typically in the format of `mail.linchi.com`.
`-q <Quiet time>`	Specifies a period of time in the format of hh-hh where hh is hours between 0 and 24, inclusive. No alerts will be sent between these two hours.
`-h`	When specified, the script only prints the usage information.

For example, you could schedule to run this script once every five minutes. In this example, whenever the script `alertErrorlog.pl` runs, it scans the errorlog file in the folder `\\SQL1\e$\mssql\mssql$apollo\log`. If it finds a critical error in the errorlog and it's between 7 A.M. and 10 P.M., the script notifies the alert recipient, which is a pager at `4321557@myTel.com` specified with argument `-r`.

The script sends the alert message in an email using the SMTP server `mail.linchi.com` specified with the argument `-m`. To the recipient, the message appears to be sent by `sql@linchi.com`, which is introduced with the argument `-a`.

The script `alertErrorlog.pl` reads the previously saved status from the file `d:\dba\log\status.log`, which is specified with the argument `-s`. The script then updates the status with the current information from the errorlog and saves the status to be used next time around.

Now it's time to discuss the code.

Examining the alertErrorlog.pl Script

The script `alertErrorlog.pl` in Listing 11-1 scans a single errorlog for errors of severity level 17 or higher.

Listing 11-1. Monitoring SQL Server Errorlog: The Basic Version

```
use strict;
use SQLDBA::Utility qw( dbaTimeDiff dbaSMTPSend dbaTime2str dbaStr2time
                        dbaSaveRef dbaReadSavedRef dbaIsBetweenTime );
use Getopt::Std;

Main: {
   my %opts;
   getopts('S:a:e:r:i:s:m:q:h', \%opts);

   # check mandatory switches
   if ( $opts{'h'} or
        !defined $opts{e} or !defined $opts{r} or !defined $opts{s} or
        !defined $opts{a} or !defined $opts{m} or !defined $opts{S} ) {
        printUsage();
   }
   my $configRef = {  # put the command-line arg's in a more readable hash
         QuietTime            => $opts{q},
         Errorlog             => $opts{e},
         SenderAccount        => $opts{a},
         StatusFile           => $opts{s},
         SMTPServer           => $opts{m},
         SQLErrAlertInterval  => $opts{i},
         DBAPager             => $opts{r},
         SQLInstance          => $opts{S}
   };

   # read saved status from the status file, if any
   my $statusRef = (-T $configRef->{StatusFile}) ?
                          dbaReadSavedRef($configRef->{StatusFile}) : {};

   my $ref = { Config => $configRef,
               Status => $statusRef };

   # Check the errorlog for critical errors
   $ref = scanErrorlog($ref);

   #  Decide whether to send alert on critical errors
```

```
38.    $ref = alertErrorlog($ref);
39.
40.    # Save status to the status file
41.    dbaSaveRef($configRef->{StatusFile}, $ref->{Status}, 'ref');
42. } # Main
43.
44. ###################
45. sub scanErrorlog {
46.    my($ref) = shift or die "***Err: scanErrorlogs() expects a reference.";
47.
48.    my $statusRef = $ref->{Status};
49.
50.    # Remove old entries recorded in the status data structure
51.    # For now, if the entry is more than 24 hours old, it is old.
52.    foreach my $errType (keys %{$statusRef->{SQLErr}}) {
53.       if ((time() - $statusRef->{SQLErr}->{$errType}->{Time})
54.               > 3600*24) {
55.          delete $statusRef->{SQLErr}->{$errType};  # remove from the hash
56.          next;
57.       }
58.       # if the entry is not that old, reset its status to good
59.       $statusRef->{SQLErr}->{$errType}->{OK} = 1;
60.    }
61.    # these two variables help shorten the expressions
62.    my $errorlog = $ref->{Config}->{Errorlog};
63.    my $server = $ref->{Config}->{SQLInstance};
64.
65.    # Now open the errorlog file and check for errors
66.    unless (open(LOG, "$errorlog")) {
67.        my $msg = "Msg: could not open $errorlog on $server.";
68.        $statusRef->{SQLErr}->{FileOpen} = {
69.                       OK       => 0,    # FileOpen is no good
70.                       ErrMsg   => $msg,
71.                       Time     => time(),
72.                       TimeStr  => dbaTime2str()
73.                     };
74.        return $ref;
75.    }
76.
77.    # Now we scan the errorlog file for regular SQL Server errors in
78.    #   the format of Error <#>, Severity <#>
79.    my ($logTime, $logTimeStr);
80.
81.    while (<LOG>) {
```

```
      # to match yyyy-mm-dd hh:mm:ss.mmm or yyyy/mm/dd hh:mm:ss.mmm
      ($logTimeStr) = /^\s*([\d\/\-]+\s+[\d\.\:]+)\s+/;
      # skip it if can't match the date/time string of the following formats
      next if ($logTimeStr !~ /\d\d(\/|-)\d\d(\/|-)\d\d\s+\d\d:\d\d:\d\d/);
      # covert to epoch seconds
      $logTime = dbaStr2time($logTimeStr);

      # skip the log entries already read by a previous run of the script
      #   as remembered in the status file
      next if ($logTime <= $statusRef->{ErrorlogLastReadTime});

      # if it's a regular SQL error, read in the next line
      if (/Error\s*\:\s*(\d+)\s*\,\s*Severity\s*\:\s*(\d+)/i) {
         my ($error, $severity) = ($1, $2);
         my $msg;

         # get the next line since it contains the actual error message text
         $_ = <LOG>;
          /^\s*[^\s]+\s+[^\s]+[^\s]+\s+(.+)$/ and ($msg = $1);

         $msg = "Err $error, $severity on $server at $logTimeStr. $msg";
         if ($severity >= 17) { # handle the error if severity >= 17
            $statusRef->{SQLErr}->{"$error\_$severity"} = {
                                 OK       => 0,          # not OK
                                 ErrMsg   => $msg,
                                 Time     => $logTime,   # epoch seconds
                                 TimeStr  => $logTimeStr # date/time string
                              };
         }
      }
   }
   close(LOG);
   # record the date/time of the last entry so that next time any entry older
   # than this date/time is skipped.
   $statusRef->{ErrorlogLastReadTime} = $logTime;
   $statusRef->{ErrorlogLastReadTimeStr} = $logTimeStr;

   return $ref;
}  # readErrorlog

#################
sub alertErrorlog {
   my($ref) = shift or die "***Err: alertErrorlog() expects a reference.";
   my ($now, $nowHour, $nowStr) = (time, (localtime)[2],dbaTime2str(time));
```

```
126.   my $msg;
127.   # for now, only one recipient. But can add more
128.   my @recipients = ($ref->{Config}->{DBAPager});
129.
130.   # Rule for alerting SQL Errors
131.   #   if 1. there is a critical SQL error
132.   #      2. SQL error was last alerted SQLErrAlertInterval minutes ago
133.   #      3. it's not in the SQLErr quiet time
134.   #   then send alert
135.
136.   foreach my $errType (keys %{$ref->{Status}->{SQLErr}}) {
137.      next if $ref->{Status}->{SQLErr}->{$errType}->{OK};   # no problem
138.      my $errRef = $ref->{Status}->{SQLErr}->{$errType};
139.
140.      # check alert time interval threshold
141.      if ( ($now - $errRef->{LastAlertedTime})
142.                     > 60*$ref->{Config}->{SQLErrAlertInterval} and
143.          !dbaIsBetweenTime($ref->{Config}->{QuietTime}) ) {
144.
145.             # send alert for this error
146.             $errRef->{SendAlertOK} = 0;  # default to bad unless proven good
147.             if ( dbaSMTPSend($ref->{Config}->{SMTPServer},
148.                              \@recipients,
149.                              $ref->{Config}->{SenderAccount},
150.                              undef,
151.                              $errRef->{ErrMsg} )) {  # send alert in header
152.                  # record alert date/time in epoch seconds and date string
153.                  $errRef->{LastAlertedTime} = time();
154.                  $errRef->{LastAlertedTimeStr} = dbaTime2str(time);
155.                  $errRef->{SendAlertOK} = 1; # alert sent successfully
156.
157.                  printf " ***%s Sent to %s; %s\n", dbaTime2str(),
158.                             $ref->{Config}->{DBAPager}, $errRef>{ErrMsg};
159.             }
160.      }
161.      $ref->{Status}->{SQLErr}->{$errType} = $errRef;
162.   }
163.   return $ref;
164. } # alertErrorlog
165.
166. ##################
167. sub printUsage {
168.    print << '--Usage--';
169. Usage:
```

```
170.  perl alertErrorlog1.pl  [-h] -a <sender account> -e <errorlog>
171.                               -i <interval> -r <pager address>
172.                               -s <status log> -q <quiet time>
173.                               -m <SMTP server> -S <SQL Server instance>
174.        -h  print this usage info
175.        -a  Sender account
176.        -e  path to the SQL Server errorlog
177.        -i  the minimum time interval between two alerts on the same error
178.        -r  alert recipient
179.        -s  status log file to save the current status of the errorlog scan
180.        -q  quiet time in hh-hh format where hh is from 0 to 24
181.        -m  SMTP mail server
182.        -S  SQL Server instance
183. --Usage--
184.    exit;
185. } # printUsage
```

This section now highlights the overall structure of the script in Listing 11-1 because you'll see it repeatedly in this chapter.

The main body of the script is structured as follows:

1. The script first organizes the command-line arguments with a more readable hash record referenced by `$configRef` on line 16. It would be cumbersome to refer to each command-line argument with a separate variable.

2. On line 28, the script reads the previously saved status information, if any, into the data structure `$statusRef` with the SQLDBA::Utility function `dbaReadSavedRef()`.

3. On line 31, the script merges the two data structures—`$configRef` and `$statusRef`—into a single hash structure under the reference `$ref`.

4. On line 35, the data structure `$ref` is passed to the function `scanErrorlog()`, which scans the errorlog with the help of the configuration options in `$ref->{Config}` and updates the status hash values in `$ref->{Status}`.

5. On line 38, the function `alertErrorlog()` examines the data structure `$ref` with the updated status information and decides whether to send an alert. If an alert is deemed necessary, the function sends the alert with a call to the function `dbaSMTPSend()` on line 147.

6. Finally, the SQLDBA::Utility function `dbaSaveRef()` saves the data structure `$ref->{Status}` to the text file, specified with the command-line option `-s`.

Using a single reference to the configuration data structure and the status data structure makes clear the logic of the script. Because a reference-based Perl data structure can arbitrarily and dynamically grow and shrink, a reference can point to any Perl data structure. As a result, the monitoring scripts in this chapter all use such a single-reference approach at the top level of the scripts to keep their overall flow consistent, nearly identical, and succinct.

Introducing the Status Data Structure

The script needs to know what has already taken place so that it can make intelligent decisions on where in the errorlog to start a new scan and whether to send a notification. More specifically, the following information should be persisted before the script exits to help guide the next invocation of the script:

- The date/time string of the last scanned entry in the errorlog
- The date/time when the last alert was sent successfully for each error

Instead of persisting only these values, it's easier and more useful to persist the entire data structure used by the script `alertErrorlog.pl` in Listing 11-1 to record the current status of the errorlog monitoring. This is possible because of the two SQLDBA::Utility functions, `dbaSaveRef()` and `dbaReadSavedRef()`. The former saves an arbitrary Perl data structure to a text file, and the latter reads from the text file the saved data structure and assigns it to a reference. If you're not familiar with these two functions, you may want to review the sections "Persisting a Data Structure to a File" and "Reading from a Persisted Data Structure" in Chapter 3.

Listing 11-2 is an example of the data structure that stores the status of monitoring a SQL Server errorlog (slightly edited for to fit the page width for readability).

Listing 11-2. The Status Data Structure Used by `alertErrorlog.pl`

```
$ref->{Status} = {
   ErrorlogLastReadTime => '1028950362',
   ErrorlogLastReadTimeStr => '2002-08-09 23:32:42.45'
   SQLErr => {
      '50000_17' => {
         OK => '0',
         ErrMsg => 'Err 50000, 17 on SQL1 at 2002-08-09 23:32. Test message.',
         Time    => '1028950362'
         TimeStr => '2002-08-09 23:32:42.45',
         SendAlertOK => 1,
         LastAlertedTime    => '1028950366',
         LastAlertedTimeStr => '2002/08/09 23:32:46 ',
      },
      '1105_17' => {
         OK => '0',
         ErrMsg => 'Err 1105, 17 on SQL1 at 2002-08-09 23:32. Could not ...',
         Time    => '1028950362'
         TimeStr => '2002-08-09 23:32:42.45',
         SendAlertOK => 1,
         LastAlertedTime    => '1028950366',
         LastAlertedTimeStr => '2002/08/09 23:32:46 ',
      },
      '50000_19' => {
         OK => '0',
         ErrMsg => 'Err 50000, 19 on SQL1 at 2002-08-09 23:32. Test message.',
         Time    => '1028950362'
         TimeStr => '2002-08-09 23:32:42.45',
         SendAlertOK => 1,
         LastAlertedTime    => '1028950366',
         LastAlertedTimeStr => '2002/08/09 23:32:46 ',
      },
   },
};
```

There are three keys underneath `$ref->{Status}`. Besides the time of the last entry in the errorlog (`ErrorlogLastReadtime` and `ErrorlogLastReadtimeStr`), `SQLErr` references the data structure that records the necessary details of each error found in the errorlog.

The script relies on `ErrorlogLastReadTime` to avoid repeatedly considering the log entries that have already been scanned when the script was previously invoked. The logic is implemented on line 91 in Listing 11-1:

```
91.       next if ($logTime <= $statusRef->{ErrorlogLastReadTime});
```

One level deeper in the nested data structure underneath `$ref->{Status}->{SQLErr}` is a hash key for each SQL Server error encountered during the errorlog scan. The code between lines 103 and 110 in Listing 11-1 populates the data structure at this level. As you can see, the concatenation of the error number and the severity level is used as the hash key to uniquely identify the error. In the example shown in Listing 11-2, there are three errors. For each error, a hash with the keys described in Table 11-2 are recorded or updated.

Table 11-2. The Keys of the Hash That Records the SQL Server Errors

ATTRIBUTE	DESCRIPTION
`OK`	A value of 0 indicates that an entry for the error is found during the current scan. A value of 1 indicates that no critical error is found during the current scan and that the hash record for the error, if any, is carried over from a previous run of the script, and it's not updated during the current scan.
`ErrMsg`	The message to be sent in the alert. The message includes the error number, the severity level, the SQL Server instance name, the date/time of the error, and the error message text.
`Time`	Date/time of the error as recorded in the errorlog. The format is epoch seconds—the number of non-leap seconds since 00:00:00 on January 1, 1970 GMT.
`TimeStr`	Date/time of the error as recorded in the errorlog in the readable format of YYYY-MM-DD hh:mi:ss.mm.
`SendAlertOK`	Value 1 indicates that the alert for this error is sent successfully, and 0 indicates otherwise.
`LastAlertTime`	Date/time when the alert is sent in the epoch seconds.
`LastAlertTimeStr`	Date/time when the alert is sent in the format of YYYY-MM-DD hh:mi:ss.mm.

NOTE *Each of the date/time related values in the status data structure is recorded in two formats: the epoch seconds—the number of non-leap seconds since 00:00:00 on Jan. 1, 1970 Greenwich mean time (GMT)—and a readable string in the format of YYYY-MM-DD hh:mi:ss.mm. This is a practice I use in most of my scripts to avoid constantly having to convert back and forth between the two formats. I sacrifice a little extra space for a gain in programming convenience and readability—a worthwhile tradeoff.*

As mentioned, the hash key for an error (for instance, error 1105) is the concatenation of its error number and its severity level (`1105_17` in this case), and the hash record for this error (referenced by `$ref->{Status}->{SQLErr}->{1105_17}` in this case) is updated whenever such an error is found during the scan of the errorlog. Because the script scans the errorlog in the chronological order, only the information from the last occurrence of the error is kept. This is precisely desirable because it guarantees that the DBA is notified of this particular error at most once during each scan.

Note that as more errors are found in the errorlog, the number of hash records under `$ref->{Status}->{SQLErr}` will increase. For troubleshooting and review purposes, you don't want to immediately remove all the information for an error from the hash even if no new occurrence of this error is found during the next scan of the errorlog. However, you don't want to keep the information forever either. In this script, the entry for an error is removed if it was last logged more than 24 hours ago. The following code segment in the function `scanErrorlog()` implements this logic:

```
50.    # Remove old entries recorded in the status data structure
51.    # For now, if the entry is more than 24 hours old, it is old.
52.    foreach my $errType (keys %{$statusRef->{SQLErr}}) {
53.       if ((time() - $statusRef->{SQLErr}->{$errType}->{Time})
54.              > 3600*24) {
55.          delete $statusRef->{SQLErr}->{$errType};  # remove from the hash
56.          next;
57.       }
58.       # if the entry is not that old, reset its status to good
59.       $statusRef->{SQLErr}->{$errType}->{OK} = 1;
60.    }
```

Meeting the Monitoring Requirements

The script `alertErrorlog.pl` in Listing 11-1 implements the following notification rule between lines 141 and 143:

```
If    (1) there's a new SQL Server error matching the configured criticality
          criteria (any error with severity level 17 or higher)
      (2) SQL error was last alerted SQLErrAlertInterval minutes ago
      (3) it's not in the quiet time period
Then  send an alert
```

The quiet time period is the time in which you don't want to be alerted no matter what error is logged in the errorlog. This is useful for a SQL Server instance that's not being supported on a 24/7 basis. For such a system, you may not appreciate being paged at 3 A.M.

The script `alertErrorlog.pl` in Listing 11-1 monitors a single errorlog and thus a single SQL Server instance. To monitor multiple instances, you can create a Windows batch file to run the Perl script in a series, each for an individual errorlog, and then schedule to run the batch file at a regular interval. It's most convenient to run such a batch file on a DBA utility server that has good network connectivity to all the monitored SQL Server machines.

The Perl script `alertErrorlog.pl` gives you a fully functional and reasonably useful tool for monitoring a SQL Server instance for any critical errors in its errorlog. That's not bad for fewer than 200 lines of code—not counting the functions imported from the utility module SQLDBA::Utility. But there's no reason to count the lines of code in that module. Just like using any other Perl modules, the work is already done for you.

"But wait! You can't be serious about monitoring SQL Server errorlogs with such a simple script," you may object. Well, you're right. But then this chapter hasn't claimed that this script is robust and comprehensive. It's a mere start to familiarize you with the basic issues of monitoring the SQL Server errorlog. The next section builds on this script and presents a robust and comprehensive script for monitoring multiple SQL Server errorlogs in a coherent manner.

Monitoring SQL Server Errorlogs: The Robust Version

> **SCENARIO**
> *You want to monitor multiple SQL Server errorlogs not only for SQL Server errors with high severity level but also for any messages deemed critical. In addition, you want to be able to accommodate future changes in monitoring requirements with little or no need to alter the script itself.*

Let's first examine where the script in Listing 11-1 may have failed to adequately address the SQL Server errorlog monitoring requirements in a typical SQL Server enterprise environment.

Investigating the Drawbacks of the Script alertErrorlog.pl

The following are the key drawbacks of the script `alertErrorlog.pl` in Listing 11-1.

There needs to be more configurations, less hard coding. What's considered a critical exception is hard-coded in the script `alertErrorlog.pl` to be any error with severity level of 17 or higher. This isn't always desirable. Not all errors with severity 17 or higher are worth knowing at 3 A.M. In addition, as you gain more experience monitoring your SQL Server installations, you'll become more specific about the errors about which you want to be notified. You definitely don't want to modify the script every time you have gained new insight into your monitoring requirements.

There's a need to monitor the size of an SQL Server errorlog. An excessively large errorlog is itself a critical exception. It makes more sense to alert the DBA of the large errorlog size than open the errorlog and scan for logged errors.

There *is* a need to notify multiple alert recipients. The script `alertErrorlog.pl` in Listing 11-1 supports only a single recipient. In an enterprise environment, a group of DBAs often manage the SQL Server installations. For a given SQL Server, there may be a need to alert both its primary DBA and its backup DBA and/or send the notification to a duty pager or even an email alias for the DBA group. Another common requirement is to alert an application support person—not the DBA—when an application process fails its database job and writes an error to the errorlog.

There *is* a need to monitor multiple errorlogs. To monitor multiple SQL Server errorlogs, you can call the script multiple times from a batch file, each with different parameters for a different SQL Server errorlog. Furthermore, note that some configuration options specified on the command line aren't specific to individual errorlogs, but common to the errorlog monitoring as a whole. For instance, it would be repetitive to specify the same SMTP server for each SQL Server errorlog. Another drawback of running the script multiple times in a batch file is that there will be as many files to persist the errorlog monitoring status as there are monitored SQL Server errorlogs.

There *is* a need to alert based on pattern-matched strings. You may want to get alerted when an errorlog entry matches a specified text pattern. This is extremely useful as a catchall mechanism because it isn't realistic to expect that everything that is or isn't worth notifying is already known at the time when the script is written.

Finally, there *is* a need to use a configuration file. It becomes unwieldy to include too many parameters all on the command line.

Building a More Versatile and Robust Script for Monitoring Errorlogs

The script `monitorErrorlogs.pl` in Listing 11-3 is built on the script in Listing 11-1 and is specifically written to overcome the drawbacks identified previously.

Listing 11-3. Monitoring SQL Server Errorlogs: The Robust Version

```
use strict;
use SQLDBA::Utility qw( dbaReadINI dbaSMTPSend dbaTime2str dbaStr2time
                        dbaSaveRef dbaReadSavedRef dbaIsBetweenTime );
use Win32::ODBC;

Main: {
   my $configFile = shift or printUsage();
   (-T $configFile) or
         die "***Err: specified config file $configFile does not exist.";

   # Read config file into $configRef
   my $configRef = dbaReadINI($configFile);

   # validate config options and set defaults
   $configRef = validateConfig($configRef);

   # read from the status file, if exists
```

```
   my $statusRef = (-T $configRef->{CONTROL}->{STATUSFILE})
                ? dbaReadSavedRef($configRef->{CONTROL}->{STATUSFILE}) : {};
   # merge into a single hash
   my $ref = { Config => $configRef, Status => $statusRef };

   # initialize the status data structure, if necessary
   $ref = initializeStatus($ref);

   # Check the errorlog files for critical errors
   $ref = scanErrorlogs($ref);

   #  Send alert on critical errors
   $ref = alertErrorlogs($ref);

   # Save status to the status file
   dbaSaveRef($configRef->{CONTROL}->{STATUSFILE}, $ref->{Status}, 'ref');
} # Main

####################
sub printUsage {
    print << '--Usage--';
Usage:
  cmd>perl monitorErrorlogs.pl <Config File>
       <Config File>   file to specify config options for alerting errorlogs
--Usage--
    exit;
}

###################
sub validateConfig {
   my $ref = shift or die "***Err: validateConfigRef() expects a reference.";

   # make sure that status file is specified
   defined $ref->{CONTROL}->{STATUSFILE}
      or die "***Err: StatusFile option is not specified.";

   #  Make sure that intervals and thresholds are set
   foreach my $server (keys %$ref) {
      next if ($server =~ /^CONTROL$/i);   # skip the CONTROL section
      # Make sure that SQL errorlog is identified for each server
      $server =~ /^server\s*:/i or
         die "***Err: $server in the heading must be prefixed with Server:";
      defined $ref->{$server}->{SQLERRORLOG} or
         die "***Err: SQLErrorlog is not specified for \[$server\].";
```

```
      # intervals are all in minutes. If they are not set explicitly
      # in the config file, their default values are set here.
      $ref->{$server}->{CONFIGERRALERTINTERVAL} ||= 60;     # in minutes
      $ref->{$server}->{ERRORLOGSIZEALERTINTERVAL} ||= 120; # in minutes
      $ref->{$server}->{SQLERRALERTINTERVAL} ||= 10;        # in minutes
      $ref->{$server}->{ONPATTERNALERTINTERVAL} ||= 20;     # in minutes
      $ref->{$server}->{ERRORLOGSIZETHRESHOLD} ||= 4000;    # in KB

      # strip off KB postfix, if any. The threshold is always
      # in KB regardless of the postfix.
      $ref->{$server}->{ERRORLOGSIZETHRESHOLD} =~ s/\s*(K|KB)\s*$//i;

      # validate OnPattern regular expressions
      foreach my $op (grep /^\s*OnPattern\d*\s*$/i, keys %{$ref->{$server}}) {
         $ref->{$server}->{$op} =~ /^(.+)\s*,\s*\[([^\[]+)\]$/;
         my ($exp, $email) = ($1, $2);
         $exp = "qr$exp" if $exp =~ /^\s*\//; # if not qr// aready, add it
         eval $exp;    # evaluate the regex
         if ($@) {     # if it's not a valid regex, skip it
            print "***$exp is not a valid regular expression.\n";
            next;
         }
         else { # for a valid regex, put associated email addresses in an array
            my %addresses;  # use its keys to get unique email addresses
            map { $addresses{$_} = 1; } split /[,;\s]+/, $email;
            $ref->{$server}->{$op} = { 'REGEX' => $exp,
                                       'EMAIL' => [ keys %addresses ]
                                     }
         }
      }
   }
   return $ref;
}  # validateConfig

###################
sub initializeStatus {
   my $ref = shift or die "***Err: initializeStatus() expects a reference.";

   foreach my $server (sort keys %{$ref->{Config}}) {
      next if $server =~ /^CONTROL$/i;
      # skip the server if its monitoring is disabled
      next if $ref->{Config}->{$server}->{DISABLED} =~ /y/i;
      my ($serverName) = $server =~ /^server\s*:\s*(.+)$/i;
```

```
# reset and/or initialize the status indicators
# (assuming eveything is fine to begin with)
# This will also initialize the hash elements, if do not exist already
$ref->{Status}->{$server}->{ErrorlogOK} = 1;
$ref->{Status}->{$server}->{Exception}->{ConfigErr}->{OK} = 1;
$ref->{Status}->{$server}->{Exception}->{ErrorlogSize}->{OK} = 1;
$ref->{Status}->{$server}->{Exception}->{SQLErr}->{OK} = 1;
$ref->{Status}->{$server}->{Exception}->{OnPattern}->{OK} = 1;

# these two intermediate variables are used to shorten expressions
# in the rest of this function
my $exceptionRef = $ref->{Status}->{$server}->{Exception};
my $configRef    = $ref->{Config}->{$server};

# Remove old entries from the status data structure
# for each exception category.

# Exception category: ConfigErr
if ((time() - $exceptionRef->{ConfigErr}->{Time})
               > 3600*24*$configRef->{REMOVESAVEDSTATUSOLDERTHAN}) {
   $exceptionRef->{ConfigErr} = { OK => 1 };
}
# Exception category: ErrorlogSize
if ((time() - $exceptionRef->{ErrorlogSize}->{Time})
               > 3600*24*$configRef->{REMOVESAVEDSTATUSOLDERTHAN}) {
   $exceptionRef->{ErrorlogSize} = { OK => 1 };
}
# Exception category: SQLErr
foreach my $err (keys %{$exceptionRef->{SQLErr}}) {
   next if $err eq 'OK';
   if ((time() - $exceptionRef->{SQLErr}->{$err}->{Time})
               > 3600*24*$configRef->{REMOVESAVEDSTATUSOLDERTHAN}) {
      delete $exceptionRef->{SQLErr}->{$err};
      next;
   }
   $exceptionRef->{SQLErr}->{$err}->{OK} = 1;
}
# Exception category: OnPattern
foreach my $err (keys %{$exceptionRef->{OnPattern}}) {
   next if $err eq 'OK';
   if ((time() - $exceptionRef->{OnPattern}->{$err}->{Time})
               > 3600*24*$configRef->{REMOVESAVEDSTATUSOLDERTHAN}) {
      delete $exceptionRef->{OnPattern}->{$err};
```

```
            next;
         }
         $exceptionRef->{OnPattern}->{$err}->{OK} = 1;
      }
   }
   return $ref;
} # initializeStatus

####################
sub scanErrorlogs {
   my $ref = shift or die "***Err: scanErrorlogs() expects a reference.";

   SERVER_LOOP:
   foreach my $server (sort keys %{$ref->{Config}}) {
      next if $server =~ /^CONTROL$/i;
      next if $ref->{Config}->{$server}->{DISABLED} =~ /y/i;
      my ($serverName) = $server =~ /^server\s*:\s*(.+)$/i;
      print "Scanning $serverName errorlog ...\n";

      # these intermediate variables are used to shorten expressions. They
      # are specific to this server
      my $statusRef    = $ref->{Status}->{$server};
      my $exceptionRef = $ref->{Status}->{$server}->{Exception};
      my $configRef    = $ref->{Config}->{$server};
      my $errorlog     = $configRef->{SQLERRORLOG};

      # Errorlog size check, if the threshold is specified.
      $exceptionRef->{ErrorlogSize}->{ActualErrorlogSize}
                    = int((stat($errorlog))[7] /1024);  # in KB
      # if the actual errorlog size is greater than the errorlog size threshold,
      # record an errorlog size exception.
      if ($exceptionRef->{ErrorlogSize}->{ActualErrorlogSize}
                > $configRef->{ERRORLOGSIZETHRESHOLD}) {
         my $msg = "$errorlog on $server too big. Scan not performed.";
         $statusRef->{ErrorlogOK} = 0;             # not OK overall
         $exceptionRef->{ErrorlogSize}->{OK} = 0;  # not OK with errorlog size
         $exceptionRef->{ErrorlogSize}->{ErrMsg} = $msg; # record the msg
         $exceptionRef->{ErrorlogSize}->{Time} = time(); # in epoch seconds
         $exceptionRef->{ErrorlogSize}->{TimeStr} = dbaTime2str(); # date string
         print "  ***$msg\n";
         # if AutoCycleErrorlog not set, don't even open the errorlog
         next SERVER_LOOP if $configRef->{AUTOCYCLEERRORLOG} !~ /y/i;
      }
```

```
# Now open the errorlog file and check for errors
my $Tries = 0;
# try three times. If still can't open the errorlog, then give up.
while (!open(LOG, "$errorlog")) {
   if (++$Tries < 3) { sleep(2); next; } # sleep for 2 seconds, and retry
   else {
      my $msg = "$Tries tries. Couldn't open $errorlog on $server.";
      $statusRef->{ErrorlogOK} = 0;
      $exceptionRef->{ConfigErr}->{OK} = 0;
      $exceptionRef->{ConfigErr}->{ErrMsg} = $msg;
      $exceptionRef->{ConfigErr}->{Time} = time();
      $exceptionRef->{ConfigErr}->{TimeStr} = dbaTime2str();
      print "  ***$msg\n";
      next SERVER_LOOP;  # can't open, then quit trying this errorlog
   }
}

# Now we scan the errorlog file
my ($time, $timeStr, $error, $severity, $msg);
while (<LOG>) {
   $statusRef->{MajorVersion}=6 if (/Microsoft\s+SQL\s+Server\s+6/i);
   ($timeStr) = /^\s*([\d\/\-]+\s+[\d\.\:]+)\s+/;
   # skip the line it if it cannot match the leading datetime string
   next if ($timeStr !~ /\d\d(\/|-)\d\d(\/|-)\d\d\s+\d\d:\d\d:\d\d/);
   $time = dbaStr2time($timeStr);

   # skip the log entries already read by a previous run
   next if $time <= $statusRef->{ErrorlogLastReadTime};
   # skip the entries older than IgnoreEntryOlderThan minutes
   next if (time() - $time) > 60*$configRef->{IGNOREENTRYOLDERTHAN};

   # if it's a regular SQL error, we need to read in the next line
   if (/Error\s*\:\s*(\d+)\s*\,\s*Severity\s*\:\s*(\d+)/i) {
      ($error, $severity)  = ($1, $2);
      # get the next line since it contains the actual error message
      $_ = <LOG>;
      /^\s*[^\s]+\s+[^\s]+[^\s]+\s+(.+)$/ and ($msg = $1);
      $_ = "Error: $error, Severity: $severity, $msg";
   }

   # checking OnPattern exceptions with matchOnPattern()
   if (my $emailAddressRef = matchOnPattern($server, $ref, $_)) {
      chomp($_);
      my $errMsg = "$timeStr OnPattern on $serverName " . $_;
```

```
            $statusRef->{ErrorlogOK} = 0;
            $exceptionRef->{OnPattern}->{OK} = 0;
            $exceptionRef->{OnPattern}->{$msg}->{Time} = $time;
            $exceptionRef->{OnPattern}->{$msg}->{TimeStr} = $timeStr;
            $exceptionRef->{OnPattern}->{$msg}->{ErrMsg} = $errMsg;
            $exceptionRef->{OnPattern}->{$msg}->{EmailAddress}
                                              = $emailAddressRef;
            print "  ***$errMsg\n";
            next;  # perform no more check on this log entry
         }

         # checking SQLErr exceptions
         if (/Error\s*\:\s*(\d+)\s*\,\s*Severity\s*\:\s*(\d+)/i) {
            my ($error, $severity) = ($1, $2);
            my $errMsg = "$timeStr Err: $error, Severity: $severity " .
                         "on $server. $msg";
            # skip it if this error # is on the exclude list
            next if ($configRef->{EXCLUDESQLERRORS} =~ /\b$error\b/i);

            #  skip it if it's on the severity include list
            if ($configRef->{INCLUDESQLSEVERITIES} =~ /\b$severity\b/i) {
               $statusRef->{ErrorlogOK} = 0;
               $exceptionRef->{SQLErr}->{OK} = 0;
               my $sqlErr = "$error\_$severity";
               $exceptionRef->{SQLErr}->{$sqlErr}->{Time} = $time;
               $exceptionRef->{SQLErr}->{$sqlErr}->{TimeStr} = $timeStr;
               $exceptionRef->{SQLErr}->{$sqlErr}->{ErrMsg} = $errMsg;
               print "  ***$errMsg\n";
            }
         }
      } # while
      close(LOG);
      $statusRef->{ErrorlogLastReadTime} = $time;
      $statusRef->{ErrorlogLastReadTimeStr} = $timeStr;
   }
   return $ref;
}  # scanErrorlogs

######################
sub matchOnPattern {
   my ($server, $ref, $msg) = @_;
   ($server && $ref && $msg) or
         die "***Err: matchOnPattern() expects a server, a reference," .
                       " and the message body.";
```

```
   my %addresses;
   my $configRef = $ref->{Config}->{$server};

   foreach my $op (grep /^\s*OnPattern\d*\s*$/i, keys %$configRef) {
      my $re = eval $configRef->{$op}->{REGEX}; # already validated
      if ($msg =~ /$re/) {
         map {$addresses{$_} = 1;} @{$configRef->{$op}->{EMAIL}};
      }
   }
   my @emails = keys %addresses;
   scalar @emails ? return [ @emails ] : return undef;
} # matchOnPattern

####################
sub alertErrorlogs {
   my $ref = shift or die "***Err: alertrrorlogs() expects a reference.";
   my ($server, $msg);
   my ($nowHour, $nowStr) = ((localtime)[2], dbaTime2str());

   foreach $server (sort keys %{$ref->{Status}}) {
      # these two intermediate variables are used to shorten
      # expressions in the rest of this chapter
      my $statusRef = $ref->{Status}->{$server};
      my $configRef = $ref->{Config}->{$server};

      next if ($configRef->{DISABLED} =~ /y/i);
      next if $statusRef->{ErrorlogOK};    # if there is no problem

   # Rule for alerting a config error
   # if 1. config error alert is enabled,
   #    2. there is a config error,
   #    3. it's not ConfigErr quiet time, and
   #    4. Last config error alerted ConfigErrorAlertInterval minutes ago
   # then  send ConfigErr alert
      my $configErrRef = $statusRef->{Exception}->{ConfigErr};
      if ( $configRef->{ALERTCONFIGERR} =~ /y/i and
           $configErrRef->{OK} == 0 and
          !dbaIsBetweenTime($configRef->{CONFIGERRQUIETTIME} ) and
          ((time() - $configErrRef->{LastAlertedTime})
                        > ($configRef->{CONFIGERRALERTINTERVAL})*60 ) ) {
            $ref = sendAlert($server, $ref, 'ConfigErr');
            next;
      }
```

```
# Rule for alerting an Errorlog Size error
# if  1. ErrorlogSize alert is enabled
#     2. Actual errorlog size > errorlog size therhold
#     3. ActualErrorlogSize > Errorlog Size Threshold
#     4. Last ErrorlogSize alerted ErrorlogSizeAlertInterval minutes old
#     5. it is not ErrorlogSize quiet time
# then  send Errorlog Size Alert
   my $errorlogSizeRef = $statusRef->{Exception}->{ErrorlogSize};
   if ($configRef->{ALERTERRORLOGSIZE} =~ /y/i and
       $errorlogSizeRef->{OK} == 0 and
       $errorlogSizeRef->{ActualErrorlogSize}
                 > $configRef->{ERRORLOGSIZETHRESHOLD}  and
       (time() - $errorlogSizeRef->{LastAlertedTime})
                 > 60*$configRef->{ERRORLOGSIZEALERTINTERVAL} and
        !dbaIsBetweenTime($configRef->{ERRORLOGSIZEQUIETTIME}) ) {
           $ref = sendAlert($server, $ref, 'ErrorlogSize');
   }

# Rule for alerting on text patterns
# if    1. OnPatternAlert is enabled,
#       2. there is a error message matching a pattern
#       3. it's not OnPatternQuietTime
#       4. This error was last alerted OnPatternInterval minutes ago
# then  send alert
   my $onPatternRef = $statusRef->{Exception}->{OnPattern};
   if ($configRef->{ALERTONPATTERN} =~ /y/i and
       !$onPatternRef->{OK} and
       !dbaIsBetweenTime($configRef->{ONPATTERNQUIETTIME})) {
      foreach my $err (keys %{$onPatternRef}) {
         next if $err =~ /^OK$/i;
         next if $onPatternRef->{$err}->{OK};

         if ((time() - $onPatternRef->{$err}->{LastAlertedTime})
                 > 60*$configRef->{ONPATTERNALERTINTERVAL}) {
            $ref = sendAlert($server, $ref, 'OnPattern', $err);
         }
      }
   }

# Rule for alerting SQL Errors
# if    1. SQLErr alert is enabled
#       2. there is a critical SQL error
#       3. it's not SQLErr quiet time
#       4. This error was last alerted SQLErrAlterInterval minutes ago
```

```
   # then  send alert
      my $SQLErrRef = $statusRef->{Exception}->{SQLErr};
      if ($configRef->{ALERTSQLERROR} =~ /y/i and
          !$SQLErrRef->{OK} and
          !dbaIsBetweenTime($configRef->{SQLERRQUIETTIME})) {
         # There might be multiple critical errors. Loop through them
         # all to check.
         foreach my $err (keys %{$SQLErrRef}) {
            next if $err =~ /^OK$/i;
            next if $SQLErrRef->{$err}->{OK};

            if ( (time() - $SQLErrRef->{$err}->{LastAlertedTime})
                          > 60*$configRef->{SQLERRALERTINTERVAL} ) {
               $ref = sendAlert($server, $ref, 'SQLErr', $err);
            }
         }
      }

   # Rule for cycling errorlog
   # if    1. AutoCycleErrorlog is enabled
   #       2. ActualErrorlogSize > Errorlog Size Threshold
   #       3. It's not SQL6.5
   # then  cycle the errorlog
      if ( ($configRef->{AUTOCYCLEERRORLOG} =~ /y/i) &&
          ($statusRef->{Exception}->{ErrorlogSize}->{ActualErrorlogSize}
                     > $configRef->{ERRORLOGSIZETHRESHOLD})  &&
           $statusRef->{MajorVersion} != 6 ) {
                 $ref = cycleErrorlog($server, $ref);
      }
   }
   return $ref;
} # alertErrorlogs

###############
sub sendAlert {
   my($server, $ref, $type, $err) = @_;

   # $type must one of ConfigErr, ErrorlogSize, SQLErr, OnPattern
   unless ($type =~ /^(ConfigErr|ErrorlogSize|SQLErr|OnPattern)$/i) {
      print "***Err: $type not among ConfigErr,ErrorlogSize,SQLErr,OnPattern\n";
      return $ref;
   }

   my $configRef = $ref->{Config};
```

```
my $statusRef = $ref->{Status}->{$server}->{Exception}->{$type};
my @recipients;
if ($type eq 'OnPattern') {
   @recipients = @{$statusRef->{$err}->{EmailAddress}};
}
else { # add DBAPAGER for this server to the list
   @recipients = split (/[;,\s]+/, $configRef->{$server}->{DBAPAGER});
   # if there is no recipient, use the DUTYPAGER in the Control section
   @recipients = ($configRef->{CONTROL}->{DUTYPAGER}) if !@recipients;
}

# send alerts for OnPattern or SQLErr messages
if ($type =~ /^(OnPattern|SQLErr)$/i ) {
   my $errRef = $statusRef->{$err};
   $errRef->{SendAlertOK} = 0;
   if (dbaSMTPSend($configRef->{CONTROL}->{SMTPSERVER},
               \@recipients,
               $configRef->{CONTROL}->{SMTPSENDER},
               undef,                    # msg body
               $errRef->{ErrMsg})) {     # msg header
      $errRef->{LastAlertedTime} = time(); # record alert sent time
      $errRef->{LastAlertedTimeStr} = dbaTime2str();
      $errRef->{SendAlertOK} = 1;          # record a successful send

      # record the sent alert in the alert log file for posterity
      my $logFile = $configRef->{CONTROL}->{ALERTLOGFILE};
      if (open (LOG, ">>$logFile")) {
         printf LOG "%s  %s  %s Sent to %s\n", dbaTime2str(),
                    $type, $errRef->{ErrMsg}, join(',', @recipients);
         close(LOG);
      }
      else { print "***Failed to open $logFile\n"; }
   }
   $ref->{Status}->{$server}->{Exception}->{$type}->{$err} = $errRef;
}

# Send alerts for ConfigErr or ErrorlogSize messages
if ($type =~ /^(ConfigErr|ErrorlogSize)$/i ) {
   my $typeRef = $ref->{Status}->{$server}->{Exception}->{$type};
   $typeRef->{SendAlertOK} = 0;
   if (dbaSMTPSend($configRef->{CONTROL}->{SMTPSERVER},
               \@recipients,
               $configRef->{CONTROL}->{SMTPSENDER},
               undef,                        # msg body
```

```
                  $typeRef->{ErrMsg})) {         # msg header
         $typeRef->{LastAlertedTime} = time();    # record alert sent time
         $typeRef->{LastAlertedTimeStr} = dbaTime2str();
         $typeRef->{SendAlertOK} = 1;             # record a successful send

         # record in the alert log file for posterity
         my $logFile = $configRef->{CONTROL}->{ALERTLOGFILE};
         if (open (LOG, ">>$logFile")) {
            printf LOG "%s  %s  %s Sent to %s\n", dbaTime2str(),
                       $type, $typeRef->{ErrMsg}, join(',', @recipients);
            close(LOG);
         }
         else { print "***Failed to open $logFile\n"; }
      }
      $ref->{Status}->{$server}->{Exception}->{$type} = $typeRef;
   }
   return $ref;
}  # sendAlert

###################
sub cycleErrorlog {
   my($server, $ref) = @_;
   ($server && $ref) or
      die "***Err: cycleErrorlog() expects server and reference.";

   my $conn;
   my $cycleRef = $ref->{Status}->{$server}->{Exception}->{CycleErrorlog};
   my ($serverName) = $server =~ /^server\s*:\s*(.+)$/i;
   my $connStr = "Driver={SQL Server};Server=$serverName;" .
                 "Trusted_Connection=Yes;database=master";

   # connect to the server to execute DBCC errorlog
   unless($conn = new Win32::ODBC ($connStr)) {
     $cycleRef->{OK} = 0;
     $cycleRef->{ErrMsg} =
           "ODBC failed to connect to $server. " . Win32::ODBC::Error();
     $cycleRef->{CycleErrorlogTimeStr} = dbaTime2str();
     $cycleRef->{CycleErrorlogTime} = time();
     $ref->{Status}->{$server}->{Exception}->{CycleErrorlog} = $cycleRef;
     return $ref;
   }

   # execute DBCC errorlog to recycle the errorlog
   if ($conn->Sql( 'DBCC errorlog' )) {
```

```
      $cycleRef->{OK} = 0;
      $cycleRef->{ErrMsg} =
             "ODBC couldn't execute DBCC ERRORLOG. " . Win32::ODBC::Error();
      $cycleRef->{CycleErrorlogTimeStr} = dbaTime2str();
      $cycleRef->{CycleErrorlogTime} = time();
    }
    else {
      1 while $conn->FetchRow();
      $cycleRef->{OK} = 1;
      $cycleRef->{ErrMsg} = "Errorlog on $server cycled";
      $cycleRef->{CycleErrorlogTimeStr} = dbaTime2str();
      $cycleRef->{CycleErrorlogTime} = time();

      if (open (LOG, ">>$ref->{Config}->{CONTROL}->{ALERTLOGFILE}")) {
         printf LOG "%s  %s\n", dbaTime2str(), $cycleRef->{ErrMsg};
         close(LOG);
      }
      else {
         print "***Err: cnnot open $ref->{Config}->{CONTROL}->{ALERTLOGFILE}\n";
      }
    }
    $ref->{Status}->{$server}->{Exception}->{CycleErrorlog} = $cycleRef;
    return $ref;
}  # cycleErrorlog
```

Wow! The script in Listing 11-3 is considerably longer than any other scripts you've seen so far in this book. However, given what it accomplishes, it isn't as complex as its length might otherwise suggest. Let's dissect the script from several perspectives: what it monitors, the configuration options it permits, its main control flow, and its key data structures.

Later, the "Setting the Options for Pattern-Matched Exceptions" section looks at the pattern-matched exceptions in more detail and gives an example to show how you can further customize the script.

Choosing What to Monitor

You can group the SQL Server exceptions monitored by this script into four categories:

- Exceptions caused by inappropriate configurations of the monitoring script itself.

- Exceptions caused by excessively large errorlog size.

- Exceptions caused by SQL Server errors.

- Exceptions as the result of matching string patterns in the errorlog. The exceptions in the same category are detected and notified with the same logic.

Understanding the Configuration Exceptions

These aren't really SQL Server error conditions. Instead, they're incorrect configurations in the configuration file passed to the script. For the script in Listing 11-3, the only configuration exception that may result in an alert to the DBA is when the errorlog path is incorrectly specified—for example, if the path points to a nonexistent directory. Whenever the script can't open an errorlog after three consecutively failed attempts, it records a configuration exception. The following code fragment from the function `scanErrorlogs()` shows how this is done:

```
my $Tries = 0;
while (!open(LOG, "$errorlog")) {
   if (++$Tries < 3) { sleep(2); next; } # sleep for 2 seconds and re-try
   else {
      my $msg = "$Tries tries. Couldn't open $errorlog on $server.";
      $statusRef->{ErrorlogOK} = 0;
      $exceptionRef->{ConfigErr}->{OK} = 0;
      $exceptionRef->{ConfigErr}->{ErrMsg} = $msg;
      $exceptionRef->{ConfigErr}->{Time} = time();
      $exceptionRef->{ConfigErr}->{TimeStr} = dbaTime2str();
      print "  ***$msg\n";
      next SERVER_LOOP;  # can't open, then quit trying this errorlog
   }
}
```

Note that the script can't tell whether an errorlog is specified correctly except that it can't open the file,[1] and failure to open the errorlog file may not necessarily be caused by an incorrectly specified errorlog path. Any one of numerous networking problems can prevent an errorlog from being opened from where the script is running.

1. This is a bit of a fib because the script could try to verify the errorlog path by checking the corresponding registry entry remotely on SQL Server. However, this introduces dependency on accessing the remote registry.

Understanding the Errorlog Size Exception

If an errorlog becomes larger than a specified size threshold, the script records an errorlog size exception. The following code fragment from the function `scanErrorlogs()` checks the errorlog size and records an errorlog size exception, if necessary:

```
# Errorlog size check, if the threshold is specified.
$exceptionRef->{ErrorlogSize}->{ActualErrorlogSize}
               = int((stat($errorlog))[7] /1024);  # actual size in KB
# compare with the size threshold
if ($exceptionRef->{ErrorlogSize}->{ActualErrorlogSize}
           > $configRef->{ERRORLOGSIZETHRESHOLD}) {
   my $msg = "$errorlog on $server too big. Scan not performed.";
   $statusRef->{ErrorlogOK} = 0;
   $exceptionRef->{ErrorlogSize}->{OK} = 0;
   $exceptionRef->{ErrorlogSize}->{ErrMsg} = $msg;
   $exceptionRef->{ErrorlogSize}->{Time} = time();
   $exceptionRef->{ErrorlogSize}->{TimeStr} = dbaTime2str();
   print "  ***$msg\n";
   # if AutoCycleErrorlog not set, don't even open the errorlog
   next SERVER_LOOP if $configRef->{AUTOCYCLEERRORLOG} !~ /y/i;
}
```

In addition to notifying the DBA of the excessive errorlog size, you can configure the script to automatically recycle the errorlog when it reaches the threshold size.

Understanding the SQL Server Error Exceptions

These are the regular SQL Server errors, each of which is identified with an error number and a severity level. The code fragment from the function `scanErrorlog()` that scans for regular SQL Server errors is as follows:

```
# if it's a regular SQL error, we need to read in the next line
if (/Error\s*\:\s*(\d+)\s*\,\s*Severity\s*\:\s*(\d+)/i) {
     ($error, $severity)  = ($1, $2);
     # get the next line since it contains the actual error message
     $_ = <LOG>;
     /^\s*[^\s]+\s+[^\s]+[^\s]+\s+(.+)$/ and ($msg = $1);
     $_ = "Error: $error, Severity: $severity, $msg";
}
```

Understanding the Pattern-Matched Exceptions

This feature allows the script to monitor the errorlog for entries that match any valid regular expression you may specify in the configuration file. Note that the errorlog entries matching a regular expression aren't necessarily exceptions in the sense that they're indicative of some sort of undesirable problems. Rather, this feature gives you ability to alert on any event, as long as an entry with an identifiable text pattern is logged in the errorlog for that event—without making any change to the script code. You'll learn more about pattern-matched exceptions in the "Setting the Options for Pattern-Matched Exceptions" section.

Setting the Configuration Options

The script in Listing 11-3 accepts a single command-line parameter—the name of a configuration file where, not surprisingly, you specify all the configuration options. Assuming the configuration file is `config.txt` in the current directory, you can run the script as follows:

```
cmd>perl monitorErrorlogs.pl config.txt
```

What the script in Listing 11-3 accomplishes will become much clearer once you've looked at the options you can specify in a configuration file. Listing 11-4 is a sample configuration file for the errorlogs of SQL Server instances, SQL1 and SQL1\APOLLO.

Listing 11-4. Sample Configuration Options for Two Errorlogs

```
[Control]
DutyPager=74321@myTel.com
SMTPServer=mail.linchi.com
SMTPSender=sql@linchi.com
AlertLogFile=e:\dba\alertErrorlog\Alert.log
StatusFile=e:\dba\alertErrorlog\Status.log

[Server:SQL1]
SQLErrorlog=\\SQL1\D$\MSSQL\LOG\Errorlog
Disabled=no
DBAPager=75114@myTel.com
IgnoreEntryOlderThan=300
RemoveSavedStatusOlderThan=2
```

```
AlertConfigErr=yes
ConfigErrAlertInterval=50
ConfigErrQuietTime=23-7

AlertErrorlogSize=yes
ErrorlogSizeThreshold=3000
ErrorlogSizeAlertInterval=120
ErrorlogSizeQuietTime=18-8

AlertSQLErr=yes
IncludeSQLSeverities=17,18,19,20,21,22,23,24,25
ExcludeSQLErrors=1608,17832,17824
SQLErrAlertInterval=5
SQLErrQuietTime=22-7

OnPattern01=/BACKUP failed.+database (SMD|RMD)\s/i, [75114@myTel.com]
OnPattern02=/Fatal error in database (?!(pubs|NorthWind)\b)/i,[73056@myTel.com]
OnPatternQuietTime = 22-7
OnPatternAlertInterval = 2

[Server:SQL1\APOLLO]
SQLErrorlog=\\SQL1\E$\MSSQL\MSSQL$APOLLO\LOG\Errorlog
Disabled=no
DBAPager=74321@myTel.com
IgnoreEntryOlderThan=300
RemoveSavedStatusOlderThan=2

AlertConfigErr=yes
ConfigErrAlertInterval=60
ConfigErrQuietTime=12-7

AlertErrorlogSize=yes
ErrorlogSizeThreshold=3000
ErrorlogSizeAlertInterval=120
ErrorlogSizeQuietTime=18-8

AlertSQLErr=yes
IncludeSQLErrors=
IncludeSQLSeverities=17,18,19,20,21,22,23,24,25
ExcludeSQLErrors=1608,17832,17824
SQLErrAlertInterval=5
SQLErrQuietTime=22-7
```

Understanding the Overall Configuration Options

The `Control` section includes the options that aren't particular to any individual SQL Server errorlog. Rather, they apply to the overall working of the script. Alerts for a SQL Server errorlog will be sent to the address of `DutyPager` if no pager address is specified with the option `DBAPager` under the section for that specific SQL Server errorlog. In other words, `DutyPager` in the `Control` section is used as a convenient fallback or default pager. The following two lines of code from the function `sendAlert()` show how the alert recipients are compiled from these options:

```
@recipients = split (/[;,\s]+/, $configRef->{$server}->{DBAPAGER});
@recipients = ($configRef->{CONTROL}->{DUTYPAGER}) if !@recipients;
```

The option `SMTPServer` specifies the address of the SMTP server that handles all the outgoing notification emails. `SMTPSender` identifies the sender of all the SMTP emails. `AlertLogFile` specifies the location of the file in which the script will record an entry after it has successfully sent an alert. This log file gives the DBA a history of the alerts sent by the script and is useful for troubleshooting purposes.

Finally, the script persists the errorlog monitoring status to the file specified by the option `StatusFile`. The information in the status file helps the script `monitorErrorlogs.pl` make better decisions when it runs again. You'll see the status data structure for the errorlog monitoring later in the "Exploring the Status Data Structure" section.

Understanding the General Errorlog Options

Underneath the section heading for a SQL Server instance such as `[Server:SQL1]`,[2] the configuration options fall into five categories. The first category of options isn't related to any particular type of exceptions. Five options are in this category:

SQLErrorlog: This option informs the script which SQL Server errorlog to open and scan. A filename using Uniform Naming Convention (UNC) is expected for this option so that the script can always find the errorlog no matter where it may be running. Because given a SQL Server instance you can easily find the location of its errorlog, you may be wondering why the script `monitorErrorlogs.pl` even requires this option. I have chosen to explicitly specify the errorlog file so that there's no dependency on having access to the remote registry where the location of the SQL Server errorlog is stored.

2. Note that the `Server:` prefix is required before each SQL Server instance name in the section heading so that the `[Control]` section heading can be reserved for the overall configurations without clashing with a server that happens to be named Control.

Disabled: This is an option to enhance the usability of the script. By setting this option to `yes`, you can conveniently exclude an errorlog from being monitored while keeping its configuration values intact in case you need to later turn on the monitoring again. If this option is set to `yes` for an errorlog, the `scanErrorlogs()` function skips the errorlog with this regular expression matching:

```
next if $ref->{Config}->{$server}->{DISABLED} =~ /^y/i;
```

DBAPager: This identifies the email address to which the script sends notifications for this particular SQL Server errorlog. This can be any email address—not necessarily that of a pager. If a list of comma-separated email addresses is specified, the notification is sent to each address on the list.

IgnoreEntryOlderThan=300: This tells the script to ignore any errorlog entries recorded more than 300 minutes ago. This essentially says that as far as monitoring errorlogs is concerned, you don't care about any entries if they are that old. The option is useful when there's no previously saved status information to tell the script where to start scanning in the errorlog, such as when the script is run for the first time.

RemoveSavedStatusOlderThan=2: This tells the script to remove from the saved status file any exception entries that are older than two days. The function `initializeStatus()` uses this option to help initialize the status hash structure—which will be discussed shortly—when the script `monitorErrorlogs.pl` starts.

Each of the other four configuration categories specifies options for a particular type of exception.

Setting the Options for Configuration Exceptions

For configuration exceptions, you can set three options:

- `AlertConfigErr=yes` enables the script to monitor configuration exceptions. If this is set to `no`, the script ignores configuration exceptions.

- `ConfigErrAlertInterval=50` instructs the script to send an alert on a configuration exception only when the previous alert for a configuration exception was sent more than 50 minutes ago.

- `ConfigErrQuietTime=23-7` says that between 23:00 and 07:00, the script should suppress any alert caused by a configuration exception.

The script monitorErrorlogs.pl also uses these three options in the notification rule for alerting a configuration exception. The following code fragment in the function alertErrorlogs() implements this configuration rule:

```
# Rule for alerting a config error
# if 1. config error alert is enabled,
#    2. there is a config error,
#    3. it's not ConfigErr quiet time, and
#    4. Last config error alerted ConfigErrorAlertInterval minutes ago
# then  send ConfigErr alert
   my $configErrRef = $statusRef->{Exception}->{ConfigErr};
   if ( $configRef->{ALERTCONFIGERR} =~ /y/i and
       $configErrRef->{OK} == 0 and
        !dbaIsBetweenTime($configRef->{CONFIGERRQUIETTIME}) and
       (time() - $configErrRef->{LastAlertedTime})
                     > ($configRef->{CONFIGERRALERTINTERVAL})*60 ) {
          $ref = sendAlert($server, $ref, 'ConfigErr');
          next;
   }
```

The other three categories have similar options, and they're used similarly to implement their respective notification rules in the function alertErrorlogs(). Instead of belaboring what are essentially the same options for each category, the subsequent sections only highlight the unique options.

Setting the Options for the Errorlog Size Exception

For the errorlog size exceptions, you can set a size threshold. For instance, with ErrorlogSizeThreshold=3000, the script doesn't record any exception unless the errorlog is larger than 3,000 kilobytes.

Setting the Options for the SQL Server Error Exception

For the regular SQL Server errors with error numbers and severity levels, the script monitorErrorlogs.pl accepts two options to explicitly include or exclude certain groups of errors:

- ExcludeSQLErrors=1608,17832,17824 tells the script to ignore the errors with these error numbers regardless of their severity levels.

- IncludeSQLSeverities=17,18,19,20,21,22,23,24,25 instructs the script to alert only on the errors whose severity levels are on the list.

Both these options help reduce the number of nuisance alerts that may be caused by harmless SQL Server errors. The script implements these options in the function `scanErrorlogs()` as follows:

```
1.  # skip it if this error is on the exclusion list
2.  next if ($configRef->{EXCLUDESQLERRORS} =~ /\b$error\b/i);
3.
4.  #  if it's on the severity list
5.  if ($configRef->{INCLUDESQLSEVERITIES} =~ /\b$severity\b/i) {
6.     $statusRef->{ErrorlogOK} = 0;
7.     $exceptionRef->{SQLErr}->{OK} = 0;
8.     my $sqlErr = "$error\_$severity";
9.     $exceptionRef->{SQLErr}->{$sqlErr}->{Time} = $time;
10.    $exceptionRef->{SQLErr}->{$sqlErr}->{TimeStr} = $timeStr;
11.    $exceptionRef->{SQLErr}->{$sqlErr}->{ErrMsg} = $errMsg;
12.    print "  ***$errMsg\n";
13. }
```

Note that on line 2 and line 5 in this code fragment, the Perl regular expression symbol `\b` matches at the word boundary. This ensures that an error number such as 1702 captured from the errorlog doesn't match 702.

Why isn't there an option to exclude a SQL Server severity level or an option to include a SQL Server error? Bear in mind that there are a limited number of severity levels in total, so there's no need to have both an exclusion option and an inclusion option. For the error numbers, experience shows that for severity levels greater than 16, only a few error numbers should be explicitly ignored.

Setting the Options for Pattern-Matched Exceptions

Finally, for pattern-matched exceptions, you can set a list of `OnPattern` options. These options are distinguished from one another by postfixing each with a sequence number. The script `monitorErrorlogs.pl` doesn't care what these sequence numbers are as long as they're unique. Furthermore, you can specify as many of these `OnPattern` options as you want. Let's examine the two `OnPattern` options in Listing 11-4:

```
OnPattern01=/BACKUP failed.+database (SMD|RMD)\s/i, [75114@myTel.com]
OnPattern02=/Fatal error in database (?!(pubs|Northwind)\b)/i,[73056@myTel.com]
```

An OnPattern option has two parts—a Perl regular expression and a list of comma-separated email addresses enclosed in a pair of square brackets. Any errorlog entry matching the pattern causes an alert to be sent to each of the email addresses listed immediately after the regular expression. If an errorlog entry matches more than one OnPattern regular expression, an alert goes to all the email addresses associated with the matched patterns. If multiple errorlog entries match an OnPattern regular expression, only the last one is included in the alert.

In the preceding example, a backup failure message for the database SMD or the database RMD—and only for these two databases—would trigger an alert to be sent to the address 75114@myTel.com. Similarly, if a message containing the string Fatal error in database is written to the errorlog and pubs or Northwind doesn't immediately follow the string, the regular expression in OnPattern02 matches and consequently will trigger an alert to 73056@myTel.com.

Examining the Flow of the Script

The main body of the script monitorErrorlogs.pl in Listing 11-3 consists of the following steps.

The script first reads the configurations from the file into a data structure referenced by $configRef, and then it validates with the function validateConfig() that the key configuration options are properly specified.

If the status file exists, the script reads the previously saved errorlog monitoring status from the status file into the data structure $statusRef. This happens with the SQLDBA::Utility function dbaReadSavedRef() in the following code fragment from the main body of the script:

```
# read from the status file
my $statusRef = (-T $configRef->{CONTROL}->{STATUSFILE})
              ? dbaReadSavedRef($configRef->{CONTROL}->{STATUSFILE}) : {};
```

After both the configuration data structure and the status data structure are merged under a single reference $ref, the script calls the function initializeStatus() to initialize the status data structure for each exception category under each enabled SQL Server instance and removes status entries that are older than the threshold specified with the option RemoveSavedStatusOlderThan in the configuration file.

Next, the function scanErrorlogs() loops through all the errorlogs listed in the configuration file to scan for the specified exceptions and update the data structure $ref->{Status} accordingly along the way. The function scanErrorlogs() is where the bulk of the work happens.

When all the errorlogs have been scanned, the script now has the updated status data structure with the information on the exceptions found in the errorlogs. The script then calls the function `alertErrorlogs()` to examine the status data structure to do the following:

1. Decide whether any alerts should be sent
2. Actually send the alerts, if any
3. Update the status data structure to record the status of sending the alerts

Finally, the script saves the updated status data structure back to the status file by calling the SQLDBA::Utility function `dbaSaveRef()`, overwriting any content already in the file. The status file is now ready to be read when the script runs the next time.

Understanding the Two Key Data Structures

The key to understanding the script `monitorErrorlogs.pl` in Listing 11-3 is understanding these two data structures:

- The configuration data structure
- The status data structure

The former stores the configuration values read from the configuration file. Once populated, this data structure doesn't change and is carried throughout the script as `$ref->{Config}`. The latter—referenced as `$ref->{Status}`—is updated with the information read from the errorlogs. It's also updated to reflect the current notification status.

Exploring the Configuration Data Structure

Listing 11-5 is a sample of the configuration data structure for the configuration options in Listing 11-4. It shows only the `Control` section and the section for the SQL Server SQL1.

Listing 11-5. A Sample Configuration Data Structure

```
$ref->{Config} = {
    CONTROL => {
        DUTYPAGER    => '74321@myTel.com',
        SMTPSERVER   => 'mail.linchi.com',
        SMTPSENDER   => 'sql@linchi.com',
        ALERTLOGFILE => 'e:\\dba\\alertErrorlog\\Alert.log',
        STATUSFILE   => 'e:\\dba\\alertErrorlog\\Status.log'
    },
    'SERVER:SQL1' => {
       SQLERRORLOG => '\\\\SQL1\\e$\\mssql\\mssql\\log\\errorlog',
       DISABLED    => 'no',
       DBAPAGER    => '75114@myTel.com',
       IGNOREENTRYOLDERTHAN       => 600,
       REMOVESAVEDSTATUSOLDERTHAN => 2,

       ALERTCONFIGERR         => 'yes',
       CONFIGERRALERTINTERVAL => 60,
       CONFIGERRQUIETTIME     => '12-7',

       ALERTERRORLOGSIZE         => 'yes',
       ERRORLOGSIZETHRESHOLD     => 3000,
       ERRORLOGSIZEALERTINTERVAL => 120,
       ERRORLOGSIZEQUIETTIME     => '18-8',

       ALERTSQLERR          => 'yes',
       INCLUDESQLSEVERITIES => '17,18,19,20,21,22,23,24,25',
       EXCLUDESQLERRORS     => '1608,17832,17824',
       SQLERRALERTINTERVAL  => 5,
       SQLERRQUIETTIME      => '22-7',

       ONPATTERN01 => '/BACKUP failed.+database (SMD|RMD)\s/i,
                                              [75114@myTel.com]',
       ONPATTERN02 => '/Fatal error in database (?!(pubs|NorthWind)\b)/i,
                                              [73056@myTel.com]',
       ONPATTERNALERTINTERVAL => 2,
       ONPATTERNQUIETTIME     => '22-7'
    }
};
```

This data structure almost mirrors the options in Listing 11-4. If you've read other chapters of this book, you should already be familiar with this data structure, produced with the SQLDBA::Utility function `dbaReadINI()`. The configuration data structure is generic across scripts. The status data structure is more interesting—and better characterizes the script `monitorErrorlogs.pl`.

Exploring the Status Data Structure

Listing 11-6 is a sample of the status data structure for the SQL Server SQL1 at the time when the script finishes executing the function `alertErrorlogs()`.

NOTE *This listing has been edited slightly for formatting purposes. Long strings, referred to by* `ErrMsg`*, are wrapped around to fit the page width. Also, the ellipse (...) is used in place of the real message text to keep the string short.*

Listing 11-6. A Sample Status Data Structure for SQL1

```
$ref->{Status}->{'SERVER:SQL1'} = {
    ErrorlogOK => '0',
    ErrorlogLastReadTime    => '1029178730',
    ErrorlogLastReadTimeStr => '2002-08-12 14:58:50.51',
    Exception => {
        SQLErr => {
            OK => '0',
            '15430_19' => {
                ErrMsg => '2002-08-12 14:58:50 Err: 15430, Severity: 19 on
                              SQL1. Limit exceeded for number of servers.',
                Time    => '1029178730'
                TimeStr => '2002-08-12 14:58:50.51',
                SendAlertOK => 1,
                LastAlertedTime    => '1029178738',
                LastAlertedTimeStr => '2002/08/12 14:58:58',
            },
            '1105_17' => {
                ErrMsg => '2002-08-12 14:58:50 Err: 1105, Severity: 17 on
                                    SQL1. Could not allocate space for ...',
                Time    => '1029178790'
                TimeStr => '2002-08-12 14:59:50.32',
                SendAlertOK => 1,
```

```
                LastAlertedTime    => '1029178791',
                LastAlertedTimeStr => '2002/08/12 14:59:51',
            }
         },
        OnPattern => {
            OK => '1',
         },
        ErrorlogSize => {
            OK => 1
            ActualErrorlogSize => 10,
         },
        ConfigErr => {
            OK => 1
         }
    }
};
```

Let's examine this data structure in more detail. The hash record for each server—for example, `$ref->{Status}->{'SERVER:SQL1'}` for SQL1—has four keys:

`ErrorlogOK => '0'` indicates that the script has detected one or more exceptions during the scan. This is an overall status indicator. When the script loops through the servers to decide whether to send an alert in the function `alertErrorlogs()`, this indicator makes it easy to skip those servers where the script didn't find any exception, in other words, with the setting `ErrorlogOK => '1'`.

`ErrorlogLastReadTime` records the date/time—in epoch seconds—of the last scanned entry in the errorlog. Next time, when the script runs, it skips all the entries in the errorlog that are older than this date/time. For instance, in Listing 11-6, only entries with a date/time later than 2002-08-12 14:58:50.51—which is the value of `ErrorlogLastReadTime`—are considered new.

`ErrorlogLastReadTimeStr` contains the same information as `ErrorlogLastReadTime` does but in a readable string format.

`Exception` references yet another hash record representing the exceptions found. In this hash, the script stores information necessary for alerting these exceptions.

Underneath the `Exception` hash key (for example, `$ref->{Status}->{'SERVER:SQL1'}->{Exception}` in Listing 11-6) are four keys representing the four exception categories:

- `ConfigErr`
- `ErrorlogSize`
- `SQLErr`
- `OnPattern`

When the script detects an exception in any of these four categories—configurations, errorlog size, SQL Server errors, and pattern-matched exceptions, the value of the `OK` hash key for that category is set to 0 along with the recorded error message and the time stamp. In Listing 11-6, the script has detected an error 15430 with severity level 19 and an error 1105 with severity 17 in the errorlog on SQL1. It therefore records the two SQL Server error exceptions shown in Listing 11-7.

Listing 11-7. A Sample Data Structure for SQL Server Error Exceptions

```
$ref->{Status}->{SQL1}->{Exception}->{SQLErr} = {
           OK => '0',
           '15430_19' => {
               ErrMsg => '2002-08-12 14:58:50 Err: 15430, Severity: 19 on
                              SQL1. Limit exceeded for number of servers.',
               Time    => '1029178730'
               TimeStr => '2002-08-12 14:58:50.51',
               SendAlertOK => 1,
               LastAlertedTime    => '1029178738',
               LastAlertedTimeStr => '2002/08/12 14:58:58',
           }
           1105_17 => {
               ErrMsg => '2002-08-12 14:58:50 Err: 1105, Severity: 17 on
                                  SQL1. Could not allocate space for ...',
               Time    => '1029178790'
               TimeStr => '2002-08-12 14:59:50.32',
               SendAlertOK => 1,
               LastAlertedTime    => '1029178791',
               LastAlertedTimeStr => '2002/08/12 14:59:51',
           }
  };
```

The value of the hash key OK in Listing 11-7 is set to 0 to flag that a SQL Server exception is detected. This flag makes it easy to loop through the exception categories because the script needs only check the value of this key for each category without having to examine any more details. In this case, there's no exception detected for any of the other three categories, thus the value of their respective hash key OK is set to 1 by default.

When the function alertErrorlogs() applies the notification rule for SQL Server error exceptions and an alert is sent, the status data structure for SQL1 is updated to include the status of sending the alert (SendAlertOK) and the date/time when the alert is sent (LastAlertedTime and LastAlertedTimeStr).

Understanding Pattern-Matched Exceptions

Among the four exception categories monitored by the script monitorErrorlogs.pl in Listing 11-3, the pattern-matched exceptions are particularly interesting. This is a powerful feature because it allows a DBA to be notified of any errorlog entry that matches an arbitrary regular expression. Now, let's find out how the script monitorErrorlogs.pl implements this feature.

Note that there's always a danger that the specified pattern might not be a valid regular expression. You absolutely do *not* want an invalid regular expression to cause a run-time error that crashes your script. Furthermore, you don't want to repeatedly contend with an invalid pattern when matching every errorlog entry.

To this end, the function validateConfig() in Listing 11-3 includes code to verify whether an OnPattern option includes a valid regular expression and removes from the configuration data structure any OnPattern option that's not valid. The validateConfig() function also updates the configuration data structure so that the regular expression and the corresponding email address for a valid OnPattern are stored explicitly as separate hash values instead of being embedded in a string, as they are originally specified in the configuration file. For example, the data structure for the OnPattern options in Listing 11-4 now looks like the following—in this case, both patterns are valid regular expressions:[3]

```
$ref->{'Config'}->{'SERVER:SQL1'}->{'ONPATTERN01'} = {
                  REGEX => 'qr/BACKUP failed.+database (SMD|RMD)\s/i',
                  EMAIL => [ 75114@myTel.com ]
  };
```

3. It would have been more optimal to store the compiled regular expressions under the hash key REGEX in the example. Unfortunately, the module Data::Dumper doesn't support the special reference returned by the quote regex operator qr//. Even though these scripts don't use Data::Dumper directly, I prefer to keep Data::Dumper available because it's such a useful tool for troubleshooting nested data structures and thus a good choice for storing the uncompiled regular expression strings.

```
$ref->{'Config'}->{'SERVER:SQL1'}->{'ONPATTERN02'} = {
             REGEX => 'qr/Fatal error in database (?!(pubs|NorthWind)\b)/i',
             EMAIL => [ 73056@myTel.com ]
};
```

In the function `validateConfig()`, the script first normalizes each of the specified patterns with Perl's quote regex operator `qr//` and then evaluates the string with the Perl's built-in `eval()` function. If there's no error, then the pattern is a valid regular expression. Otherwise, it's not a valid regular expression, and the pattern is removed from the configuration data structure.

NOTE *Using the* `eval()` *function is a standard Perl technique for trapping errors that may otherwise be fatal.*

Then, in the function `scanErrorlogs()`, before it checks each errorlog entry to determine whether it's a critical SQL Server error, the script first calls the function `matchOnPattern()` to see whether the entry matches any of the specified regular expressions. If there's a match, the script sets the value of the `OK` key to 0 under `OnPattern`, for instance, `$ref->{Status}->{'SERVER:SQL1'}->{Exception}->{OnPattern}`. Listing 11-8 illustrates a sample of the data structure for recording a pattern-matched exception.

Listing 11-8. A Sample Data Structure for `OnPattern` *Exceptions*

```
$ref->{'Status'}->{'SERVER:SQL1'}->{Exception}->{'OnPattern'} = {
       'OK' => '0',
       'Backup failed for database RDM.' => {
            'ErrMsg' => '2002-08-13 00:01:08.30 OnPattern on SQL1
                         Error: 50000, Severity: 17, Backup failed for RDM.',
            'Time'    => '1029211268',
            'TimeStr' => '2002-08-13 00:01:08.30',
            'SendAlertOK' => 1,
            'LastAlertedTime'    => '1029211393',
            'LastAlertedTimeStr' => '2002/08/13 00:03:13 ',
            'EmailAddress' => [
                              '75114@myTel.com'
                            ]
       },
       'Fatal error in database RDM.' => {
```

```
            'ErrMsg' => '2002-08-13 00:01:21.39 OnPattern on SQL1
                        Error: 50000, Severity: 19, Fatal error in database RDM.',
            'Time'    => '1029211281',
            'TimeStr' => '2002-08-13 00:01:21.39',
            'SendAlertOK' => 1,
            'LastAlertedTime'    => '1029211393',
            'LastAlertedTimeStr' => '2002/08/13 00:03:13 ',
            'EmailAddress' => [ '73056@myTel.com '  ]
        }
};
```

This data structure is similar to that for SQL Server error exceptions, illustrated in Listing 11-7. But instead of using the number and the severity level of an error in the hash key, an `OnPattern` exception is identified with the message text itself. In Listing 11-8, the error messages `Backup failed for database RDM` and `Fatal error in database RDM` are the two hash keys under `$ref->{Status}->{'SERVER:SQL1'}->{Exception}->{OnPattern}`. The information recorded under these keys is similar to that recorded for a SQL Server error exception in Listing 11-7. The main difference is the presence of a list of email addresses referenced by the key `EmailAddress`. Later, the function `alertErrorlogs()` sends the `OnPattern` message—the value of the key `ErrMsg` in the same hash record—to each address on this list.

Because the message text is used as the hash key, by its very nature when the same messages are found in a scan, only the last one is recorded. Different messages in the errorlog may match the same regular expression. However, because they're different, their information will be recorded separately under different hash keys and therefore will result in different alerts.

Customizing the Script monitorErrorlogs.pl

Comprehensive as it may be in monitoring SQL Server errorlogs, the script `monitorErrorlogs.pl` in Listing 11-3 is far from complete. You can further improve the script in numerous ways. Experience has convinced me that I'll never be able to anticipate all the details of the monitoring requirements in my SQL Server environment, let alone anticipate those of your environments.

Customizing the script in Listing 11-3 most likely means adding another exception category. Let's see with an example what it takes to make the script monitor a new category of exceptions.

Assume that you'd like to be alerted when a SQL Server instance is restarted. How do you accomplish that? Not surprisingly, you can turn to the errorlog to tell when the instance is restarted. Numerous entries at the beginning of the SQL

Server errorlog are indicative of a freshly started SQL Server instance. The following is an example:

```
2002-08-18 20:53:26.74 server SQL Server is starting at priority ...
```

If the script is scheduled to run once every five minutes, you can look for this entry and check whether its time is within the last seven minutes or any number of minutes greater than five and smaller than 10. This is to guarantee that you receive at least one alert and at most two about the fact that the SQL Server instance was restarted.

To support alerting on a SQL Server restart, you can introduce a new exception category. Let's call it the *restart* category. You must make the following modifications to the script in Listing 11-3 to monitor exceptions in the restart category:

- At the minimum, the configuration file should include the following options for the restart exception: `AlertRestart`, `RestartThreshold`, and `RestartQuietTime`. Only when `AlertRestart` is `yes` will the script check for any restart exception. A restart exception is detected if the SQL Server instance was restarted less than `RestartThreshold` minutes ago. The option `RestartQuietTime` specifies the time during which no alerts of the restart exception should be sent.
- In the function `initializeStatus()`, you need to add code mimicking the initialization of the data structures for the `ConfigErr` and `ErrorlogSize` exceptions.
- In the function `scanErrorlogs()`, you also need to add code to scan for restart exceptions and populate the data structure `$exceptionRef->{Restart}` if the instance was restarted more recently than `RestartThreshold` minutes ago.
- In the function `alertErrorlogs()`, you again need to add code to implement the rule for alerting the restart exception. This code segment should mimic that for alerting a `ConfigErr` exception or an `ErrorlogSize` exception.

This chapter won't present the complete script that supports the monitoring of restart exceptions. Instead, this chapter leaves that as an exercise for you.

NOTE *If you don't care about the ability to specify special notification requirements such as* `RestartQuietTime`*, you can simply use an* `OnPattern` *regular expression to capture the errorlog string that indicates the server is starting. That way, you don't need to modify any code.*

Monitoring Drive Space Shortage

> **SCENARIO**
> *You want to be alerted when the free space on a disk drive on any of your SQL Servers drops below a critical threshold—say 200MB—so that you can take immediate measures to prevent the server from actually running out of disk space or misbehaving because of a disk space shortage.*

A disk drive running low on free space isn't on the list of exceptions outlined at the beginning of this chapter. So why is it being discussed here?

It's true that monitoring disk drives for low free space is probably considered as proactive monitoring because you're monitoring something that's not yet a problem but is likely to be a problem if the trend continues. The primary motivation for discussing it in this section is to demonstrate the advantage of reusing the infrastructure for monitoring SQL Server errorlogs.

Alternatively, you can of course write a self-contained script to scan the servers and check the free space of each disk drive on every server. If the free space of a drive is smaller than the threshold, the script sends a notification to the DBA.

However, because you've already put in place the errorlog monitoring script `monitorErrorlogs.pl` in Listing 11-3, why not take advantage of it? You can translate many monitoring problems into a problem of writing appropriate messages to the SQL Server errorlog and then let the errorlog monitoring tool pick up the errors and send the alerts.

Monitoring disk drives that are running critically low on space is one such problem.

The script `alertDriveSpace.pl` in Listing 11-9 loops through a list of SQL Servers. For each, it executes the SQL Server extended stored procedure `xp_fixeddrives`, via Open Database Connectivity (ODBC), to list the local hard drives with their respective free space. If the free space of a drive is below a threshold value, the script raises a SQL Server error of severity level 18. The error is written to the errorlog and will then be caught by the script `monitorErrorlogs.pl` when it scans that errorlog or for that matter by any tool you may be using to monitor the SQL Server errorlog.

Listing 11-9. Alerting on the Critically Low Free Drive Space

```
use strict;
use SQLDBA::Utility qw( dbaReadINI dbaTime2str dbaStr2time );
use Win32::ODBC;

Main: {
   my $configFile = shift or printUsage();

   (-T $configFile) or
      die "***Err: specified config file $configFile does not exist.";

   # Read config file into $configRef
   my $ref = dbaReadINI($configFile);

   # check free drive space for each drive and alert accordingly
   $ref = checkFreeSpace($ref);
} # Main

#####################
sub checkFreeSpace {
   my $ref = shift or die "***Err: checkFreeSpace() expects a reference.";

   foreach my $server (sort keys %$ref) { # loop through each server
      next if $server =~ /^control$/i;    # skip Control section
      next if $ref->{$server}->{DISABLED} =~ /^y/i; # skip disabled server
      # skip the server if its disk free space threshold is not properly set
      next unless $ref->{$server}->{FREESPACETHRESHOLD} =~ /^\d+/;

      # strip off the Server: prefix if any
      my ($instance) = $server =~ /^server\s*:\s*(.+)/i;
      $instance ||= $server;

      my $sql;
      my $connStr = "Driver={SQL Server};Server=$instance;" .
                           "Trusted_Connection=Yes;Database=master";
      my $db = new Win32::ODBC ($connStr);
      unless ($db) {
         print "***could not connect to server $instance.\n";
         next;
      };

      $sql = 'EXEC master..xp_fixeddrives';
      unless ($db->Sql($sql)) {
```

```
         while ($db->FetchRow()) {
           my ($d, $s) = $db->Data;
           $ref->{$server}->{FREEDRIVESPACE}->{$d} = $s;
         }
      }
      else {
         print "***executing $sql. ", Win32::ODBC::Error(), "\n";
      }

      # loop through the drives to decide whether to alert
      foreach my $drive (sort keys %{$ref->{$server}->{FREEDRIVESPACE}}) {
         if ($ref->{$server}->{FREEDRIVESPACE}->{$drive} <
                        $ref->{$server}->{FREESPACETHRESHOLD}) {
             $sql = "raiserror('$drive drive on $instance is below " .
                               $ref->{$server}->{FREESPACETHRESHOLD} .
                               "MB',18, -1) with log";
             unless ($db->Sql($sql)) {   # raise the error
                1 while $db->FetchRow();
             }
             else {
                 print "***executing $sql.\n";
             }
         }
      }

      $db->Close();
      return $ref;
   }
}  # checkFreeSpace

################
sub printUsage {
   print << '--Usage--';
Usage:
  cmd>perl alertDriveSpace.pl <Config File>

       <Config File>   Config file that specifies what servers to check
--Usage--
   exit;
} # printUsage
```

This script accepts a configuration file on the command line. The expected configuration file looks like Listing 11-10.

Listing 11-10. Configuration Options for Alerting on Low Drive Space

```
[Server:SQL1]
Disabled=no
FreeSpaceThreshold=500   # in MB

[Server:SQL2\APOLLO]
Disabled=no
FreeSpaceThreshold=200   # in MB
```

For each server identified with a section heading in the file illustrated in Listing 11-10, you can specify two options for this script: `Disabled` and `FreeSpaceThreshold`. If `Disabled` is set to `yes`, the script doesn't check the drive space on that server. `FreeSpaceThreshold`, not surprisingly, specifies the free space threshold in megabytes (MB) below which an alert will be sent. For instance, `FreeSpaceThreshold=500` under `[Server:SQL1]` indicates that the free drive space threshold is 500MB for the server SQL1. You can schedule to run the script once every few minutes as follows:

```
cmd>perl alertDriveSpace.pl config.txt
```

With the configuration file in Listing 11-10, if a drive—for example, the E drive—is found to have less than 500MB of free space on SQL1, the script `alertDriveSpace.pl` logs the following error in its SQL Server errorlog:

```
2002-08-06 21:37:56.84 spid52 Error: 50000, Severity: 18, State: 1
2002-08-06 21:37:56.84 spid52 E drive on SQL1 is below 500MB.
```

In practice, there's no need to use a separate configuration file just for this script. It's better to share a configuration file used by some other SQL Server monitoring script, such as `monitorErrorlogs.pl` in Listing 11-3. As long as the option `FreeSpaceThreshold` is specified for a server, the script `alertDriveSpace.pl` will check the free space for each drive on that server and log an error in the SQL Server errorlog if the free space is below the threshold. The script will ignore any other options in the configuration file, which may be specified and used by other scripts.

Again, this script does *not* make any decision on whether an alert should be sent or actually send any alert. That's the job of the errorlog monitoring facility. The strategy of taking advantage of the available SQL Server errorlog monitoring infrastructure helps prevent you from reinventing the wheel and therefore keeps the script short and compact. It also helps curtail the proliferation of ad-hoc monitoring setups.

NOTE *As long as you carefully control what's being written to it, the SQL Server errorlog can conveniently serve as a central repository for all the DBA-related messages. For instance, instead of having each scheduled job include a specially customized mechanism for sending alerts, they can all simply raise appropriate SQL Server errors and leave the rest of the notification work to the errorlog monitoring infrastructure. This also makes it convenient to review the job-related error messages in a single place.*

Monitoring Availability: The Basic Version

SCENARIO
You want to be notified when your SQL Server machine becomes unreachable or when the connection attempt to a SQL Server instance fails.

Listing 11-3 presented a robust and comprehensive script for monitoring errors recorded in SQL Server errorlogs. The script is only concerned with what's in the errorlog, and it's completely oblivious of whether the SQL Server instance and its various components are functioning properly or functioning at all. The fact that you don't see any error in an errorlog doesn't mean that the state of the SQL Server instance is good. Perhaps nothing is written to the errorlog because SQL Server is hung.

In addition to monitoring the errorlog, you must also check that SQL Server is available along with various services it offers.

For now, let's assume that SQL Server availability means that a client can log into the SQL Server instance. SQL Server becomes unavailable either when the machine isn't reachable over the network or when the SQL Server instance isn't accepting any connection attempt for whatever reason.

To verify that the machine is reachable, you can ping its Internet Protocol (IP) address. However, because most of the application clients connect to the SQL Server instance name instead of its IP address, it's better to ping the server name. This helps test the network name resolution in the process. To verify that SQL Server is accepting client connections, you can try to actually log into the instance. As is often said, the proof of the pudding is in the eating.

In summary, you need to perform two availability checks: a reachability check and a SQL Server connectivity check. The next section explores alerting on more problems that may cause SQL Server to become less than completely available and expands the script to perform additional availability checks.

The key design objectives for the script to monitor SQL Server availability are twofold:

- To accurately and promptly detect that SQL Server isn't available
- To sufficiently notify the DBA of the unavailability with minimal redundancy in notification

To accomplish these two objectives, the script `alertAvailability.pl` in Listing 11-11 monitors SQL Server availability as follows:

1. The script runs from a job scheduler at a regular interval—about once every five minutes.
2. The script performs two availability checks on the SQL Server instance: First, it pings the machine using the Net::Ping module, and second, if the ping is successful, it attempts to connect to the SQL Server instance via ActiveX Data Objects (ADO).
3. If one of the availability checks fails, the availability status is set to bad, and a bad status counter is incremented to record the consecutive number of times the bad status is reported.
4. If both availability checks are successful, the availability status is set to good, and the bad status counter is reset to zero.
5. If the value of the bad status counter reaches three, the script sends a notification to the DBA.
6. The script saves the data structure representing the status of the availability monitoring to a text file to be used when the script runs the next time.

Listing 11-11 shows the script `alertAvailability.pl` to monitor the availability of a SQL Server instance.

Listing 11-11. Monitoring SQL Server Availability: The Basic Version

```
use strict;
use Getopt::Std;
use Win32::OLE;
use SQLDBA::Utility qw( dbaTimeDiff dbaSMTPSend dbaTime2str dbaStr2time
                        dbaSaveRef dbaReadSavedRef dbaIsBetweenTime );
use Net::Ping;

Main: {
   my %opts;
   getopts('S:a:s:m:r:q:h', \%opts);  # get the command-line arguments

   # check mandatory switches
   if (   $opts{'h'}
       or !defined $opts{S} or !defined $opts{a} or !defined $opts{s}
       or !defined $opts{m} or !defined $opts{r} ) {
       printUsage();
   }
   # put the command-line options in a more readable hash
   my $configRef = {
         SQLInstance   => $opts{S},
         SenderAccount => $opts{a},
         StatusFile    => $opts{s},
         SMTPServer    => $opts{m},
         DBAPager      => $opts{r},
         QuietTime     => $opts{q}
   };

   # read saved status from the status file, if any
   my $statusRef = (-T $configRef->{StatusFile}) ?
                       dbaReadSavedRef($configRef->{StatusFile}) : {};

   # merge the config hash and the status hash so that later steps only
   # need to pass a single reference
   my $ref = { Config => $configRef,
               Status => $statusRef };
```

```
   # Check availability
   $ref = checkAvailability($ref);

   #  Decide whether to send an alert, and send the alert if necessary
   $ref = alertAvailability($ref);

   # Save status to the status file
   dbaSaveRef($configRef->{StatusFile}, $ref->{Status}, 'ref');
} # Main

##########################
sub checkAvailability {
   my $ref = shift or
         die "***Err: checkAvailability() expects a reference.";
   my $server = $ref->{Config}->{SQLInstance};

   CheckHeartbeat: {
      if (isPingOK($server)) {                  # first ping the server
         $ref->{Status}->{Ping}->{OK} = 1;      # ping is good
      }
      else {
         $ref->{Status}->{Ping}->{OK} = 0;      # ping is not good
         $ref->{Status}->{Ping}->{LastFailed} = dbaTime2str();
         $ref->{Status}->{BadStatusCounter}++;  # increment bad status counter
         $ref->{Status}->{ErrMsg} = 'Failed to ping ' . $server;
         last CheckHeartbeat;
      }

      if (isSQLConnectOK($server)) {             # then test SQL connection
         $ref->{Status}->{SQLConnect}->{OK} = 1; # SQL connect is good
      }
      else {
         $ref->{Status}->{SQLConnect}->{OK} = 0; # SQL connect is bad
         $ref->{Status}->{SQLConnect}->{LastFailed} = dbaTime2str();
         $ref->{Status}->{BadStatusCounter}++;   # increment bad status counter
         $ref->{Status}->{ErrMsg} = 'Failed to connect to ' . $server;
         last CheckHeartbeat;
      }
   }  # CheckHeartbeat
```

```
   if ($ref->{Status}->{Ping}->{OK} and
       $ref->{Status}->{SQLConnect}->{OK}) {  # both ping and SQL connect is good
      $ref->{Status}->{OK} = 1;               # availability check is good
      $ref->{Status}->{BadStatusCounter} = 0; # reset the bad status counter to 0
   }
   else {                                     # availability check is not good
      $ref->{Status}->{OK} = 0;
   }
   return $ref;
} # checkAvailability

####################
sub isPingOK {
    my $server = shift or
      die "***Err: IsPingOK() expects a server name.";

    $server =~ s/\\.+$//;  # remove the instance name, if any

    my $p = Net::Ping->new("icmp");   # ICMP ping
    my $r = $p->ping($server, 2);     # 2 second timeout
    $p->close();
    return $r;
} # IsPingOK

######################
sub isSQLConnectOK {
   my $server = shift or
      die "***Err: IsSQLConnectOK() expects a server name.";

   my $conn = Win32::OLE->new('ADODB.Connection');  # use ADO
   $conn->{ConnectionTimeout} = 2;                  # 2 second timeout
   $conn->Open("Driver={SQL Server};Server=$server;Trusted_Connection=yes");
   my $err = Win32::OLE->LastError();
   $conn->Close();
   $err ? return 0 : return 1;   # if there is no error, it's good
} # isSQLConnectOK

########################
sub alertAvailability {
   my $ref = shift or die "***Err: alertAvailability() expects a reference.";

   my @recipients = ($ref->{Config}->{DBAPager});  # set recipient to DBAPager
   $ref->{Status}->{AlertSent} = 0; # assume nothing is sent
```

```
   # if there is an availability problem and is not quiet time
   # send an alert
   if ($ref->{Status}->{OK} == 0 and                # availability is not good
       $ref->{Status}->{BadStatusCounter} > 2 and   # not good for > 2 times
       !dbaIsBetweenTime($ref->{Config}->{QuietTime})) { # not in quiet time
         if (dbaSMTPSend($ref->{Config}->{SMTPServer},   # send it via SMTP mail
                     \@recipients,
                     $ref->{Config}->{SenderAccount},
                     undef,
                     $ref->{Status}->{ErrMsg})) {       # send msg in the header
            $ref->{Status}->{AlertSent} = 1;             # successfully sent
            $ref->{Status}->{LastAlertSent} = dbaTime2str();
            $ref->{Status}->{BadStatusCounter} = 0;      # reset to 0

            printf "%s  %s Sent to %s\n", dbaTime2str(),
                        $ref->{Status}->{ErrMsg}, $ref->{Config}->{DBAPager};
         }
   }
   return $ref;
} # alertAvailability

##################
sub printUsage {
    print << '--Usage--';
Usage:
   cmd>perl alertAvailability.pl [-h] -S <instance>
                                      -a <sender account>
                                      -s <status file>
                                      -m <SMTP server>
                                      -r <recipient>
                                      -q <quiet time>
        -h    print this usage info
        -S    SQL Server instance
        -a    Sender account
        -s    status log file to save the current availability status
        -m    SMTP mail server
        -r    alert recipient
        -q    quiet time
--Usage--
    exit;
} # printUsage
```

If you want to monitor the SQL Server instance SQL1 and send alerts to dba@myTel.com, you can schedule to run the script alertAvailability.pl in Listing 11-11 as follows (all should be on a single command line):

```
cmd>perl alertAvailability.pl -S NJSQLO1 -a sql@linchi.com -s status.log
                              -m smtp.linchi.com -r dba@myTel.com -q 22-7
```

Table 11-3 describes all the command-line arguments accepted by the script alertAvailability.pl.

Table 11-3. Command-Line Parameters for the Script in Listing 11-11

PARAMETER	DESCRIPTION
-S <SQL Server name>	Specifies the name of the SQL Server instance to be checked for availability.
-s <Status file>	The file to which the availability monitoring status will be saved.
-r <Recipient>	The email address of the alert recipient.
-m <SMTP server>	The SMTP mail server.
-a <SMTP account>	Specifies the account of the SMTP email sender.
-q <Quiet time>	Specifies a period of time—in the format of hh-hh—in which no alerts will be sent, where hh is between 1 and 24, inclusive.
-h	When specified, the script only prints out the usage information.

The data structure used throughout the script is referenced by the variable $ref. The command-line arguments are kept in the hash record referenced by $ref->{Config}. This hash record is intuitive because its keys mirror the command-line options.

The data structure referenced by $ref->{Status} is more interesting. The script alertAvailability.pl in Listing 11-11 updates this data structure in the functions checkAvailability() and alertAvailability(). The data structure represents the status of availability monitoring. Listing 11-12 illustrates a sample of this data structure.

Listing 11-12. A Sample Status Data Structure `$ref->{Status}`

```
$ref = {Status} = {
          OK => '0',
          ErrMsg => 'Failed to connect to SQL1',
          BadStatusCounter => '2',
          AlertSent => '0',
          LastAlertSent => '2002/08/18 00:52:07 ',
          SQLConnect => {
                           OK => '0'
                           LastFailed => '2002/08/19 00:52:36 ',
          },
          Ping => {
                      OK => 1
          }
};
```

Table 11-4 describes the keys used to record the availability status of a SQL Server instance:

Table 11-4. The Keys of the Hash Record `$ref->{Status}`

KEY	DESCRIPTION
`OK`	A value of 0 indicates that one of the availability checks failed, whereas a value of 1 shows that both availability checks succeeded.
`ErrMsg`	The value is updated with the message from the latest failed availability check. This message will be sent if an alert is deemed necessary.
`BadStatusCounter`	Number of consecutive times an availability check has failed. It's reset to 0 if an availability check finds no problem.
`AlertSent`	A value of 1 indicates that an alert is sent successfully during the current run of the script, and 0 indicates no alert is sent during this run.
`LastAlertSent`	Time when the last alert is sent.
`SQLConnect`	A reference to a hash recording the status of the SQL Server connectivity check.
`Ping`	A reference to a hash recording the status of the reachability check.

Before the script exits its main body, it calls the SQLDBA::Utility function `dbaSaveRef()` to write the data structure `$ref->{Status}` to the status file—specified on the command line with parameter `-s`—for the script `alertAvailability.pl` to read when the script runs the next time.

In the status data structure `$ref->{Status}`, the only information that's required is `$ref->{Status}->{BadStatusCounter}`, which records the number of consecutive times the availability check has failed. When this fails three times, the `alertAvailability()` function sends an alert if the other alert conditions are also met. As is the case in the other scripts in this chapter, it's convenient to save and read the entire status data structure.

The function `isPingOK()` and the function `isSQLConnectOK()` perform two availability checks. The former uses the Perl module Net::Ping to test for reachability by sending an Internet Control Message Protocol (ICMP) echo message to the remote host. The following code fragment reproduced from Listing 11-11 implements the function `isPingOK()`:

```
###############
sub isPingOK {
    my $server = shift or
      die "***Err: IsPingOK() expects a server name.";

    $server =~ s/\\.+$//;  # remove the instance name, if any

    my $p = Net::Ping->new("icmp");   # ICMP ping
    my $r = $p->ping($server, 2);     # 2 second timeout
    $p->close();
    return $r;
} # IsPingOK
```

The function `isSQLConnectOK()` tries to establish a trusted connection to the SQL Server instance through ADO using the SQL Server ODBC driver.

The function `alertAvailability()` implements the following notification rule:

```
If    (1) one of the availability checks fails,
      (2) the availability checks have failed for more than two consecutive
          times, and it is not in the quiet time period
Then  send an alert.
```

It's no accident to insist that the availability check must have failed three consecutive times before an alert is sent. This number came out of using such a script in practice for an extended time. The primary motivation is to minimize the number of nuisance alerts that may be caused by a variety of network problems or hiccups without unduly prolonging the time it takes to get notified.

Monitoring Availability: The Robust Version

> **SCENARIO**
> *In addition to checking whether the server can be reached and the SQL Server instance is accepting client connections, you want to be notified when a database becomes unusable or one of the SQL Server support services becomes unresponsive on any SQL Server instances in your environment.*

Let's see how you can make the script in Listing 11-11 more robust and comprehensive for monitoring SQL Server availability.

Improving the Availability Monitoring

There's plenty of room for improvement. In a nutshell, you need to be able to monitor multiple servers, you need more comprehensive availability checking, you need the script to be more customizable, you want to minimize the number of nuisance alerts, and you want better alert logging.

Monitoring Multiple SQL Server Instances

The script `alertAvailability.pl` in Listing 11-11 monitors only one SQL Server instance. To monitor multiple SQL Server instances, you can call the script multiple times in a batch file, each monitoring a separate instance. Although this is a workable solution, it quickly becomes difficult to manage as the number of instances increases.

Enabling More Availability Checks

For a DBA, the availability of the database service to the customers is more than a server that can be pinged and a SQL Server instance that accepts client connections. You have to ensure that the databases are usable and that the SQL Server Agent is running. Optionally, you may want to check that the SQL Mail is functioning, if you support SQL Mail in your environment.

Customizing Availability Checks

You need to be able to specify which of the availability checks should be performed for a given SQL Server instance. It's reasonable to expect that you always want to maintain the server reachability and the connectivity to the SQL Server instance. But it doesn't make sense to insist that all the databases be usable on every instance. A database may be in the middle of a restore on a standby server and therefore can't be queried. Also, a database may be taken offline on purpose. In either case, you don't want to be alerted because the database isn't available.

Keeping the Number of Alerts Under Control

To avoid being flooded with alerts on the availability of the same instance, you can set a minimum alert interval so that another alert for the same instance won't be sent for at least this amount of time.

Logging Alerts

Having a history of all the alerts that the script has sent is useful in reviewing the availability of the SQL Server instances. It's also useful for troubleshooting purposes. Every time the script sends an alert, it should record an entry in a log file.

Creating a Robust Script for Monitoring Availability

The script `monitorAvailability.pl` in Listing 11-13 incorporates the previous improvements.

Listing 11-13. Monitoring SQL Server Availability: The Robust Version

```
use strict;
use Win32::OLE;
use Net::Ping;
use SQLDBA::Utility qw( dbaSMTPSend dbaTime2str dbaSaveRef dbaInStrList
                        dbaReadSavedRef dbaIsBetweenTime dbaReadINI );

Main: {
   my $configFile = shift or printUsage();
   (-T $configFile) or
         die "***Err: specified config file $configFile does not exist.";
```

```
   # Read config file into $configRef
   my $configRef = dbaReadINI($configFile);

   # validate config options and set defaults
   $configRef = validateConfig($configRef);

   # read the saved status info from the status file, if available
   my $statusRef = (-T $configRef->{CONTROL}->{STATUSFILE})
                    ? dbaReadSavedRef($configRef->{CONTROL}->{STATUSFILE})
                    : {};
   # merge the config data structure and the status data structure
   my $ref = { Config => $configRef,
               Status => $statusRef };

   # Check availability
   $ref = checkAvailability($ref);

   #  Decide whether to send an alert, and sends the alert, if necessary
   $ref = alertAvailability($ref);

   # Save status to the status file for the next invocation
   dbaSaveRef($configRef->{CONTROL}->{STATUSFILE}, $ref->{Status}, 'ref');
} # Main

###################
sub validateConfig {
   my $configRef = shift or die "***Err: validateConfig() expects a reference.";

   foreach my $server (sort keys %{$configRef}) {
      next if $server =~ /^control$/i;  # skip Control section
      next if $configRef->{$server}->{DISABLED} =~ /^y/i; # skip disabled server

      # only validate AlertInteval option (for demo purpose)
      if (!defined $configRef->{$server}->{ALERTINTERVAL} or
          $configRef->{$server}->{ALERTINTERVAL} !~ /\d+/) {
          $configRef->{$server}->{ALERTINTERVAL} = 20; # default to 20 minutes
      }
   }
   return $configRef;
} # validateConfig
```

```
######################
sub checkAvailability {
   my $ref = shift or die "***Err: checkAvailability() expects a reference.";

   foreach my $server (sort keys %{$ref->{Config}}) {
      next if $server =~ /^control$/i;  # skip Control section
      next if $ref->{Config}->{$server}->{DISABLED} =~ /^y/i; # skip the disabled

      # strip off the Server: prefix if necessary
      my ($instance) = $server =~ /^SERVER\s*:\s*(.+)$/i;
      # These two structures are to shorten the expressions
      # in the rest of the function
      my $statusRef = $ref->{Status}->{$server};
      my $configRef = $ref->{Config}->{$server};

      CHECK_HEALTH: {
         # Ping the server
         if ($configRef->{CHECKPING} =~ /^y/i and       # ping enabled
             !isPingOK($instance)) {                    # but failed to ping
            $statusRef->{Ping}->{OK} = 0;               # flag it as no good
            $statusRef->{Ping}->{LastFailed} = dbaTime2str();
            $statusRef->{BadStatusCounter}++;  # increment bad status counter
            $statusRef->{ErrMsg} = 'Failed to ping ' . $instance;
            last CHECK_HEALTH;
         }
         else {
            $statusRef->{Ping}->{OK} = 1;               # ping is good
         }
         # login to the instance
         if ($configRef->{CHECKCONNECTION} =~ /^y/i and # enabled
             !isSQLConnectOK($instance)) {              # but failed to connect
            $statusRef->{SQLConnect}->{OK} = 0;         # flag it as no good
            $statusRef->{SQLConnect}->{LastFailed} = dbaTime2str();
            $statusRef->{BadStatusCounter}++;    # increment the bad status cntr
            $statusRef->{ErrMsg} = 'Failed to connect to ' . $instance;
            last CHECK_HEALTH;
         }
         else {
            $statusRef->{SQLConnect}->{OK} = 1;        # SQL connect is good
         }
         #check DB status
         if ($configRef->{CHECKDATABASES} =~ /^y/i) {  # enabled
            my $DBStatusRef = getDBStatus($ref, $server);
            if (!$DBStatusRef->{OK}) {                 # but some db is no good
```

```
            $statusRef->{DB}->{OK} = 0;             # flag it as no good
            $statusRef->{DB}->{LastFailed} = dbaTime2str();
            $statusRef->{BadStatusCounter}++; # increment the bad status cntr
            $statusRef->{ErrMsg} = $DBStatusRef->{DBMsg};
            $statusRef->{DB}->{DBMsg} = $DBStatusRef->{DBMsg};
            last CHECK_HEALTH;
         }
         else {
            $statusRef->{DB}->{OK} = 1;   # all databases are good
         }
      }
      else {
         $statusRef->{DB}->{OK} = 1; # if not enabled, default db to good
      }
      # Check SQLAgent
      if ($configRef->{CHECKSQLAGENT} =~ /^y/i and  # enabled
          !isSQLAgentOK($instance)) {               # but not running
         $statusRef->{SQLAgent}->{OK} = 0;          # flag it as no good
         $statusRef->{SQLAgent}->{LastFailed} = dbaTime2str();
         $statusRef->{BadStatusCounter}++;  # increment bad status counter
         $statusRef->{ErrMsg} = "SQLAgent on $instance is not running";
         last CHECK_HEALTH;
      }
      else {
         $statusRef->{SQLAgent}->{OK} = 1;   # SQLAgent is good
      }
      # Check SQL Mail
      if ($configRef->{CHECKSQLMAIL} =~ /^y/i and  # enabled
          !isSQLMailOK($instance)) {               # but not working
         $statusRef->{SQLMail}->{OK} = 0;          # flag it as no good
         $statusRef->{SQLMail}->{LastFailed} = dbaTime2str();
         $statusRef->{BadStatusCounter}++;   # increment bad status counter
         $statusRef->{ErrMsg} = "SQLMail on $instance is not running";
         last CHECK_HEALTH;
      }
      else {
         $statusRef->{SQLMail}->{OK} = 1;  # SQLMail is good
      }
   }   # CHECK_HEALTH

   # Now update the overall status
   if ( $statusRef->{Ping}->{OK}       and
        $statusRef->{SQLConnect}->{OK} and
        $statusRef->{SQLAgent}->{OK}   and
```

```
            $statusRef->{SQLMail}->{OK}    and
            $statusRef->{DB}->{OK}) {
         $statusRef->{OK} = 1;                # overall status flag is good
         $statusRef->{BadStatusCounter} = 0;  # reset the bad status counter
      }
      else {
         $statusRef->{OK} = 0;  # flag the overall status as bad
      }
   }
   return $ref;
} # checkAvailability

################
sub isPingOK {
    my $server = shift or die "***Err: IsPingOK() expects a server name.";
    $server =~ s/\\.+$//;  # remove the instance name, if any

    my $p = Net::Ping->new("icmp");  # ICMP ping
    my $r = $p->ping($server, 2);    # 2 second timeout
    $p->close();
    return $r;
} # IsPingOK

######################
sub isSQLConnectOK {
   my $server = shift or die "***Err: IsSQLConnectOK() expects a server name.";

   my $conn = Win32::OLE->new('ADODB.Connection') or return 0;
   $conn->{ConnectionTimeout} = 2;   # 2 second timeout is hard coded
   $conn->Open("Driver={SQL Server};Server=$server;Trusted_Connection=yes");
   my $state = $conn->{State};
   $conn->Close();
   return $state;
} # isSQLConnectOK

####################
sub isSQLAgentOK {
   my $server = shift or die "***Err: isSQLAgentOK() expects a server name.";

   my ($serverName, $instanceName) = $server =~ /^([^\\]+)\\([^\\]+)$/i;
   my $conn = Win32::OLE->new('ADODB.Connection') or return 0;
   $conn->{ConnectionTimeout} = 2;
   $conn->Open("Driver={SQL Server};Server=$server;Trusted_Connection=yes");
```

```
   my $rc = 0;
   my $sql = q/EXEC master..xp_cmdshell 'net start'/;
   my $rs = $conn->Execute($sql);
   if ($rs) {
      while ( !$rs->{EOF} ) {
         my $info = $rs->Fields('output')->{Value};
            if (! $instanceName) {
              $rc = 1 if $info =~ /(SQLExec|SQLServerAgent)/i;
            }
            else {
              $rc = 1 if $info =~ /SQLAgent\$$instanceName/i;
            }
         $rs->MoveNext();
      }
      $rs->Close;
   }
   $conn->Close;
   return $rc if $rc;
} # isSQLAgentOK

####################
sub isSQLMailOK {
   my $server = shift or die "***Err: isSQLMailOK() expects a server name.";

   my $conn = Win32::OLE->new('ADODB.Connection') or return 0;
   $conn->{ConnectionTimeout} = 2;
   $conn->Open("Driver={SQL Server};Server=$server;Trusted_Connection=yes");

   my $rc = 0;
   my $sql = q/ DECLARE @rc int
                SET NOCOUNT ON
                CREATE TABLE #tmp (a varchar(125) null)
                INSERT #tmp EXEC @rc = master..xp_findnextmsg
                SELECT 'output' = @rc/;
   my $rs = $conn->Execute($sql);
   if ($rs) {
      while ( !$rs->{EOF} ) {
         $rc = $rs->Fields('output')->{Value};
         $rs->MoveNext();
      }
      $rs->Close;
   }
   $conn->Close();
   return !$rc;
```

```
} # isSQLMailOK

###################
sub getDBStatus {
   my ($ref, $server) = @_;  # $server is expected to have SERVER: prefix
   my $configRef = $ref->{Config}->{$server};
   my $DBStatusRef = $ref->{Status}->{$server}->{DB};

   my ($instance) = $server =~ /^SERVER\s*:\s*(.+)$/i;

   my $conn = Win32::OLE->new('ADODB.Connection');
   $conn->{ConnectionTimeout} = 2;
   my $connStr = "Driver={SQL Server};Server=$instance; ";
      $connStr .= "Database=master;Trusted_Connection=yes";
   $conn->Open($connStr);
   if (!$conn->{State}) {
      $DBStatusRef->{OK} = 0;
      $DBStatusRef->{DBMsg} = "Failed to connect to $instance";
      return $DBStatusRef;
   }

   # get the database names first
   my @DB = ();
   my $sql = q/SELECT name FROM master..sysdatabases
               WHERE name NOT IN ('pubs', 'model', 'NorthWind')
                 AND name NOT LIKE '%test%' /;

   my $rs = $conn->Execute($sql);
   if ($rs) {
      while ( !$rs->{EOF} ) {
         @DB = (@DB, $rs->Fields('name')->{Value});
         $rs->MoveNext();
      }
      $rs->Close;
   }
   else {
       $DBStatusRef->{OK} = 0;
       $DBStatusRef->{DBMsg} = "Problem executing $sql";
       return $DBStatusRef;
   }

   my $badDB = undef;
   foreach my $db (@DB) {
      $sql = qq/SET QUOTED_IDENTIFIER ON
```

```
                SELECT count(*) FROM \"$db\"..sysobjects
                 WHERE name = 'sysobjects'/;
      # construct a list of databases that cannot be queried
      unless ($rs = $conn->Execute($sql)) {
         my $dbErr = 'Msg: ' . Win32::OLE->LastError();
         # check whether Offline mode is excluded from alerting
         next if ($dbErr =~ /Database.+is\s+offline/is and
                  dbaInStrList($db, $configRef->{EXCLUDEOFFLINE}));
         # check whether Loading mode is excluded from alerting
         next if ($dbErr =~ /Database.+is\s+in.+\s+restore/is and
                  dbaInStrList($db, $configRef->{EXCLUDELOADING}));
         $badDB .= " $db";
      }
   }
   if ($badDB) {
      $DBStatusRef->{OK} = 0;
      $DBStatusRef->{DBMsg} = "Problem querying database(s):$badDB";
      return $DBStatusRef;
   }
   $conn->Close();
   $DBStatusRef->{OK} = 1;
   return $DBStatusRef;
} # getDBStatus

######################
sub alertAvailability {
   my $ref = shift or die "***Err: alertAvailability() expects a reference.";

   foreach my $server (sort keys %{$ref->{Config}}) {
      next if $server =~ /^control$/i;
      next if $ref->{Config}->{$server}->{DISABLED} =~ /^y/i;

      my ($instance) = $server =~ /^SERVER\s*:\s*(.+)$/i;
      # these two variables are to shorten the expressions
      my $statusRef = $ref->{Status}->{$server};
      my $configRef = $ref->{Config}->{$server};

      my @receivers = split (/[;,\s]+/, $configRef->{DBAPAGER});
      # if the pagers are not specified for the server, use the DutyPager
      # as the default
      @receivers = ($ref->{Config}->{CONTROL}->{DUTYPAGER}) unless @receivers;
```

```
      # if there is a problem for more than two consecutive times,
      # and it is not quiet time,
      # and it's been longer than the AlertInterval since an alert was last sent
      # send an alert
      if ($statusRef->{OK} == 0 and    # status flag is good
          $statusRef->{BadStatusCounter} > 2 and # bad status counter > 2
          !dbaIsBetweenTime($ref->{Config}->{QuietTime}) and # not quiet time
          (time() - $statusRef->{LastAlertSentTime})  # minutes since last alert
                        > ($configRef->{ALERTINTERVAL})*60 ) {

            if (dbaSMTPSend($configRef->{SMTPServer},
                        \@receivers,
                        $configRef->{SenderAccount},
                        undef,
                        $statusRef->{ErrMsg})) { # send msg in the mail header
               $statusRef->{AlertSent} = 1;       # send was good
               $statusRef->{LastAlertSentTime} = time();
               $statusRef->{LastAlertSentTimeStr} = dbaTime2str();
               $statusRef->{BadStatusCounter} = 0;  # reset bad status counter

               open(LOG, ">>$ref->{Config}->{CONTROL}->{ALERTLOGFILE}");
               printf LOG "%s  %s. Sent to %s\n", dbaTime2str(),
                              $statusRef->{ErrMsg}, $statusRef->{DBAPAGER};
               close(LOG);
            }
      }
   }
   return $ref;
} # alertAvailability

##################
sub printUsage {
    print << '--Usage--';
Usage:
   cmd>perl monitorAvailability.pl <Config File>
        <Config File>  file to specify config options for monitoring availability
--Usage--
    exit;
}
```

You can configure the behavior of the script `monitorAvailability.pl` in Listing 11-13 by setting configuration options in a configuration file. Listing 11-14 is an example for two SQL Server instances, SQL1 and SQL1\APOLLO.

Listing 11-14. A Sample Configuration File for `monitorAvailability.pl`

```
[Control]
AlertLogFile=Alert.log
StatusFile=status.log
DutyPager=74321@myTel.com
SMTPServer=mail.linchi.com
SMTPSender=dba@linchi.com

[Server:SQL1]
Disabled=no
DBAPager=71234@myTel.com

CheckPing=yes
CheckConnection=yes
CheckSQLAgent=yes
CheckDatabases= yes
CheckSQLMail=no
QuietTime=18-8
AlertInterval=20

[Server:SQL1\APOLLO]
Disabled=no
DBAPager=75114@myTel.com

CheckPing=yes
CheckConnection=yes
CheckSQLAgent=yes
CheckDatabases=yes
CheckSQLMail=yes
QuietTime=24-6
AlertInterval=15
ExcludeLoading=RDM,UDM
ExcludeOffline=Survey
```

Table 11-5 describes the options you can specify in the `Control` section of a configuration file passed to `monitorAvailability.pl`.

Table 11-5. Options in the `Control` *Section for Availability Monitoring*

OPTION	DESCRIPTION
`AlertLogFile`	Every time an alert is sent, a log entry is written to the text file specified by this option.
`StatusFile`	The script persists the status data structure for the availability monitoring to this file.
`DutyPager`	This specifies the email address. Alerts goes to this address if the `DBAPager` option for the server isn't specified.
`SMTPServer`	This is the SMTP server that all the outgoing notification messages use.
`SMTPSender`	This is the SMTP account to identify the sender of the alert messages.

Under the section heading for each SQL Server instance, you can specify the options described in Table 11-6. All options are optional.

Table 11-6. Options for Each SQL Server Instance for Availability Monitoring

OPTION	DESCRIPTION
`Disabled`	If this is set to `yes`, this SQL Server instance won't be checked for availability.
`DBAPager`	All notifications will be sent to this email address.
`CheckPing`	The script pings the server for reachability only if the option is set to `yes`.
`CheckConnection`	The script performs the connectivity check on the SQL Server instance only if the option is set to `yes`.
`CheckDatabases`	The script queries each database to verify whether the database is usable only if the option is set to `yes`.
`CheckSQLAgent`	The script tries to determine whether the `SQLServerAgent` is online only if the option is set to `yes`.
`CheckSQLMail`	The script checks whether SQL Mail is responsive only if the option is set to `yes`.

Table 11-6. Options for Each SQL Server Instance for Availability Monitoring (Continued)

OPTION	DESCRIPTION
QuietTime	This option expects two values in the format of hh-hh, where hh is between 1 and 24, inclusive. This option specifies that between these two hours, no alerts should be sent.
AlertInterval	This option expects an integer that specifies the number of minutes that must elapse before another alert on this SQL Server instance can be sent.
ExcludeLoading	This option expects a comma-separated list of database names. If any of these databases is in the loading mode, the database usability check won't flag it as unusable.
ExcludeOffline	This option expects a comma-separated list of database names. If any of these databases is offline, the database won't be flagged as unusable.

You can run the script `monitorAvailability.pl` from the command line as follows if the options are specified in the file `config.txt` in the current directory:

```
cmd>perl monitorAvailability.pl config.txt
```

Examining the Script monitorAvailability.pl

The main body of this script is nearly identical to that of the script in Listing 11-3. It consists of the following steps:

1. The script `monitorAvailability.pl` first reads the configuration options from the configuration file supplied on the command line and reads the status information—saved by the previous invocation of the script—from the status file specified with the option `StatusFile` in the configuration file. The script then combines the data structure for the configuration options and the data structure for the status information into a single data structure referenced by `$ref`. The script uses the data structure `$ref` throughout the rest of the script.

2. The script updates the status data structure `$ref->{Status}` with the function `checkAvailability()`, which performs the specified availability checks for each of the SQL Server instances listed in the configuration file.

3. Next, the function `alertAvailability()` inspects the updated `$ref` data structure, applies the notification rules to decide whether to send any alert, and actually sends the alert if one is deemed necessary. This function also updates the `$ref` data structure to record the information related to sending the alert.

4. Finally, the script calls the SQLDBA::Utility function `dbaSaveRef()` to save the status data structure `$ref->{Status}` to be used when the script runs the next time. Note that the key information to save for each SQL Server instance is the number of consecutively failed availability checks and the last time an alert is sent. These two pieces of information are used to help the script make better notification decisions.

Before moving on to a different topic, the next two sections cover two interesting issues in the design of the availability monitoring script `monitorAvailability.pl` in Listing 11-13.

Deciding What Error Conditions to Alert

First, in the function `checkAvailability()`, the code that performs the availability check resides inside a labeled block:

```
CHECK_HEALTH: {
   ...
}
```

As soon as an availability problem is detected, the script exits the `CHECK_HEALTH` code block without performing any more availability checks. It's clear that for each SQL Server instance only one availability error message is recorded and eventually sent in the alert. This raises an obvious question: What should be sent to the DBA when there are multiple availability error conditions? The function `checkAvailability()` has a simple design: The following error conditions are checked sequentially, and the first error condition encountered is sent; the rest are ignored:

1. Can the script ping the server?

2. Can the script connect to the SQL Server instance?

3. Can the script query the databases?

4. Is SQL Server Agent running?

5. Is SQL Mail running?

This has worked well in practice. But if you're wary of the danger that an important error condition is masked by a less important error, you may decide to notify the DBA of all the important errors instead of only one of them. This doesn't always make sense because there's often natural dependency among the error conditions. Obviously, when one error implies another, only the first one should trigger an alert. For example, when you can't ping a server, it's only annoying to also tell the DBA that you can't log into the SQL Server instance.[4]

However, it does make sense to send two alerts when the SQL Server Agent stops running and a database becomes unusable at the same time. Each of the two availability problems has different implications for the DBA, and there's no apparent dependency between them.

Detecting Database Availability

The second point of interest is how to detect that a database has become unusable. The work happens in the function `getDBStatus()`. To determine whether the query has run into any problem, the function checks the ADO resultset and the error condition returned by executing the following query from the master database:

```
SET QUOTED_IDENTIFIER ON
SELECT count(*) FROM "<db>"..sysobjects
WHERE name = 'sysobjects'
```

where `<db>` is the name of the database to be checked. When the SQL Server instance fails to execute this query, the resultset from the ADO `Execute` method is undefined.

Things become more interesting when you need to exclude a database that's offline or in the loading mode from triggering an alert. To avoid having to explicitly deal with various versions of SQL Server, this script inspects the error message

4. This is usually true in a TCP/IP network, which is almost universally used these days.

returned from executing the query to determine whether the error is caused by the database being offline or in the loading mode. It helps to review the following code fragment from the function `getDBStatus()`:

```
my $badDB = undef;
foreach my $db (@DB) {
   $sql = qq/SET QUOTED_IDENTIFIER ON
             SELECT count(*) FROM \"$db\"..sysobjects
              WHERE name = 'sysobjects'/;
   # construct a list of databases that cannot be queried
   unless ($rs = $conn->Execute($sql)) {
      my $dbErr = 'Msg: ' . Win32::OLE->LastError();
      # check whether Offline mode is excluded from alerting
      next if ($dbErr =~ /Database.+is\s+offline/is and
               dbaInStrList($db, $configRef->{EXCLUDEOFFLINE}));
      # check whether Loading mode is excluded from alerting
      next if ($dbErr =~ /Database.+is\s+in.+\s+restore/is and
               dbaInStrList($db, $configRef->{EXCLUDELOADING}));
      $badDB .= " $db";
   }
}
```

In this case, the script examines the error message returned from `Win32::OLE->LastError()`. Alternatively, you can inspect the ADO errors collection for the same error message patterns. Either would work. But it's easier to work with `Win32::OLE->LastError()` because there's no need to iterate through a Component Object Model (COM) collection, which is what you must do with the ADO `Errors` collection.

Monitoring SQL Server Cluster: The Basic Version

> **SCENARIO**
> *Some of your SQL Server instances are running in a Microsoft failover cluster. You want to be notified when the SQL Server cluster has experienced a significant state change.*

With the errorlog monitoring script `monitorErrorlogs.pl` in Listing 11-3 and the availability monitoring script `monitorAvailability.pl` in Listing 11-13, your toolset is comprehensive in terms of monitoring SQL Server exceptions. Well, at least that's true if you're monitoring stand-alone SQL Server instances.

When a SQL Server instance runs in a failover cluster, however, these scripts won't notify you when an instance has failed over to a different node or when a node has just gone down. There will be no critical errors in the errorlog for the errorlog monitoring script to catch, and the script `monitorAvailability.pl` won't report any outage if the instance comes up online quickly before the script runs the next time or when only one node isn't up.

Monitoring the Cluster Events

In addition to monitoring the SQL Server errorlog and the SQL Server availability, what should a DBA monitor when a SQL Server instance runs in a failover cluster? In other words, what changes in a cluster are significant enough that the DBA should be notified? This section discusses a script to monitor the following five cluster-specific events:

- The cluster isn't accessible.
- A cluster node is evicted from the cluster.
- The status of a cluster node isn't up.
- The status of a cluster group isn't online.
- A group has moved from one node to another.

Now, how do you detect these changes in a cluster? You can get information on the first four events in the previous list by simply querying the cluster. To ascertain whether a group has moved to a different node, you can resort to the tried-and-true trick of taking two snapshots of the current state of the cluster and then performing a comparison to see whether there's any difference between the two.

Choosing a Tool to Monitor the Cluster

There are two techniques to get a snapshot of the current state of the cluster nodes or groups. One is to parse the output of the command-line utility `cluster.exe`. If the name of the cluster is NYCLUSTER, after issuing the command, you'll obtain a result similar to the one shown in Listing 11-15.

Listing 11-15. Getting Cluster Node Status

```
cmd>cluster NYCLUSTER node

Listing status for all available nodes:

Node                      Node ID       Status
------------------------- ------------- ---------------------
NYCLSQLNODE1              1             Up
NYCLSQLNODE2              2             Up
```

This result is intuitive, which shows that both nodes of the cluster are up. If the status of any of the nodes isn't `Up`, the DBA should know. Likewise, in the following example, after issuing the command, you'll get a cluster group status report similar to what's shown in Listing 11-16.

Listing 11-16. Getting Cluster Group Status

```
cmd>cluster NYCLUSTER group

Listing status for all available resource groups:

Group                Node            Status
-------------------  --------------  ---------
Cluster Group        NYCLSQLNODE1    Online
SQL Server           NYCLSQLNODE1    Online
```

In this case, the cluster has two groups. They're both online and currently both running on node `NYCLSQLNODE1`. The DBA should be notified if a group status isn't `Online`.

NOTE *The command-line utility* `cluster.exe` *comes with Windows 2000. You don't need to install anything.*

Another technique to get the status of a cluster is to use the COM automation interface provided by the Cluster Automation Object. This is a more robust approach because the programming interface is better defined and better

documented than the output format of `cluster.exe`. With the latter, you have to study its various outputs to identify the text patterns, and there's no guarantee that the output format will remain the same when a new version of `cluster.exe` is released. Using the Cluster Automation Server becomes a necessity if the script needs to explore aspects of a cluster beyond simple node or group status. Although `cluster.exe` gives almost any information you want with respect to a cluster, the text output can be too messy to parse.

You'll see how to parse the output of `cluster.exe`, primarily to learn that you can take advantage of another command-line utility in your Perl scripts. The next section uses the Cluster Automation Server to retrieve information from a cluster.

Creating a Script to Monitor the Cluster Events

In addition, to highlight the key points, the script in Listing 11-17 monitors only one cluster with all the configuration options specified on the command line. This is similar to the improvements you've made to the SQL Server errorlog monitoring and SQL Server availability in the previous sections. This section is a sneak preview. The "Monitoring SQL Server Cluster: The Robust Version" section adds more features to monitor multiple SQL Server clusters and improve its robustness.

Listing 11-17 shows the script `alertCluster.pl` to monitor the events of the SQL Server cluster identified previously.

Listing 11-17. Monitoring SQL Server Cluster: The Basic Version

```
use strict;
use SQLDBA::Utility qw( dbaSMTPSend dbaTime2str dbaSaveRef dbaSetDiff
                        dbaReadSavedRef dbaSetCommon );
use Getopt::Std;

Main: {
   my %opts;
   getopts('C:s:r:a:m:', \%opts);

   ($opts{C} and $opts{a} and $opts{s} and $opts{m} and $opts{r}) or
          printUsage();

   my $configRef = {
         Cluster          => $opts{C},
         SenderAccount    => $opts{a},
         StatusFile       => $opts{s},
         SMTPServer       => $opts{m},
         DBAPager         => $opts{r}
   };
```

```
   # read saved status from the status file, if any
   my $statusRef = (-T $configRef->{StatusFile}) ?
                       dbaReadSavedRef($configRef->{StatusFile}) : {};
     # use a single reference in the rest of the script
     my $ref = { Config => $configRef,
                 SavedStatus => $statusRef };

   # get current status
   $ref = getStatus($ref);

   # check the current status and compare it with the saved one
   $ref = checkStatus($ref);

   #  Decide whether to send an alert
   $ref = alertStatus($ref);

   # Save status to the status file for the next time
   dbaSaveRef($configRef->{StatusFile}, $ref->{CurrentStatus}, 'ref');
} # Main

##################
sub printUsage {
    print << '--Usage--';
Usage:
  cmd>perl alertCluster.pl -C <cluster> -s <statusFile> -r <dbaPager>
                                        -a <senderAccount> -m <SMTPServer>

      -C <cluster>     Name of the cluster to be moitored
      -s <statusFile>  Name of the file to record the cluster status
      -r <dbaPager>    pager email address
      -a <sendAccount> Account of the sender on the SMTP server
      -m <SMTPServer>  SMTP server
--Usage--
    exit;
}

#################
sub getStatus {
   my $ref = shift or
      die "***Err: getStatus() expects a reference.";

   my $statusRef;
   my $cluster = $ref->{Config}->{Cluster};
```

```
# get node names and status with cluster node
my $msg = `cluster $cluster node`;   # backtick operator
my @status = split /[\n\r\f]+/, $msg;
# skip to the result row
1 while (shift @status) !~ /----/;  # the header

# now @status only has the result rows
unless (@status) {
   $msg =~ s/\s+/ /g;
   $msg =~ s/\s*$//;
   $statusRef->{OK} = 0;
   $statusRef->{AlertStatus}->{FailedQuery}->{Times} =
            $ref->{SavedStatus}->{AlertStatus}->{FailedQuery}->{Times} + 1;
   $statusRef->{AlertStatus}->{FailedQuery}->{Msg} =
            "cluster $cluster node. $msg.";
   $ref->{CurrentStatus} = $statusRef;
   return $ref;
}

foreach (@status) {
   my ($node, $status) = $_ =~ /^\s*(\w+)\s+\d+\s+(.+)/;
   $status =~ s/\s*$//;
   $statusRef->{ClusterStatus}->{Nodes}->{$node}->{Status} = $status;
}

# get group names and status with cluster group
$msg = `cluster $cluster group`;    # backtick operator
@status = split /[\n\r\f]+/, $msg;
# skip to the result row
1 while (shift @status) !~ /^----/; # the header

unless (@status) {
   $msg =~ s/\s+/ /g;
   $msg =~ s/\s*$//;
   $statusRef->{OK} = 0;
   $statusRef->{AlertStatus}->{FailedQuery}->{Times} =
            $ref->{SavedStatus}->{AlertStatus}->{FailedQuery}->{Times} + 1;
   $statusRef->{AlertStatus}->{FailedQuery}->{Msg} =
                       "cluster $cluster group. $msg.";
   $ref->{CurrentStatus} = $statusRef;
   return $ref;
}
```

```
   my $node_re = join('|', keys %{$statusRef->{ClusterStatus}->{Nodes}});
   foreach (@status) {
      my ($group, $node, $status) = $_ =~ /^\s*(.+)\s+($node_re)\s+(.+)/;
      $group =~ s/\s*$//;
      $statusRef->{ClusterStatus}->{Groups}->{$group}->{Status} = $status;
      $statusRef->{ClusterStatus}->{Groups}->{$group}->{Node} = $node;
   }
   $statusRef->{OK} = 1;
   $ref->{CurrentStatus} = $statusRef;
   return $ref;
}  # getStatus

##################
sub checkStatus {
   my ($ref) = shift or die "***Err: checkStatus() expects a reference.";
   my $savedRef = $ref->{SavedStatus}->{ClusterStatus};
   my $currentRef = $ref->{CurrentStatus}->{ClusterStatus};
   my $alertStatusRef;

   my @diff = ();
   my @savedNodes = keys %{$savedRef->{Nodes}};
   my @currentNodes = keys %{$currentRef->{Nodes}};
   my @savedGroups = keys %{$savedRef->{Groups}};
   my @currentGroups = keys %{$currentRef->{Groups}};

   # check if a node is not UP
   foreach my $node (@currentNodes) {
      if ($currentRef->{Nodes}->{$node}->{Status} !~ /^Up$/i) {
         $alertStatusRef->{NodeNotUp}->{Msg} .= "$node,";
         $ref->{CurrentStatus}->{OK} = 0;
      }
   }

   # check if a group is not online
   foreach my $group (@currentGroups) {
      my $status = $currentRef->{Groups}->{$group}->{Status};
      if ($status !~ /^Online$/i) {
         $alertStatusRef->{GroupNotOnline}->{Msg} .= "$group,";
         $ref->{CurrentStatus}->{OK} = 0;
      }
   }

   # check if a node is evicted
   if (@diff = dbaSetDiff(\@savedNodes, \@currentNodes)) {
```

```
            my $nodes = join(',', map {"'" . $_ ."'"} @diff);
            $alertStatusRef->{NodeEvicted}->{Msg} = $nodes;
            $ref->{CurrentStatus}->{OK} = 0;
         }

         # check if a group is moved to another node
         foreach my $group (dbaSetCommon(\@currentGroups, \@savedGroups)) {
            my $oldNode = $savedRef->{Groups}->{$group}->{Node};
            my $newNode = $currentRef->{Groups}->{$group}->{Node};
            if ($oldNode ne $newNode ) {
               $alertStatusRef->{GroupMoved}->{Msg} .= "$group: $oldNode->$newNode,";
               $ref->{CurrentStatus}->{OK} = 0;
            }
         }
         $ref->{CurrentStatus}->{AlertStatus} = $alertStatusRef;
         return $ref;
      } # checkStatus

      #################
      sub alertStatus {
          my $ref = shift or die "***Err: alertStatus() expects a reference.";
          return $ref if $ref->{CurrentStatus}->{OK};

          my $alertStatusRef = $ref->{CurrentStatus}->{AlertStatus};
          my @receivers = ( $ref->{Config}->{DBAPager} );

          foreach my $alertType (sort keys %{$alertStatusRef}) {
            next if (     $alertType eq 'AccessFailed'
                      and $alertStatusRef->{AccessFailed}->{Times} <= 2);
            # tidy up the message
            $alertStatusRef->{$alertType}->{Msg} =~ s/,$//;
            $alertStatusRef->{$alertType}->{Msg}
                              = 'Cluster ' . $ref->{Config}->{Cluster} .
                                " $alertType: " . $alertStatusRef->{$alertType}->{Msg};
            # send via SMTP
            if (dbaSMTPSend($ref->{Config}->{SMTPServer},
                        \@receivers,
                        $ref->{Config}->{SenderAccount},
                        undef,
                        $alertStatusRef->{$alertType}->{Msg})) {
               $ref->{CurrentStatus}->{AlertSent}->{$alertType}->{OK} = 1;
               $ref->{CurrentStatus}->{AlertSent}->{$alertType}->{AlertSentTimeStr}
                                 = dbaTime2str();
               printf "%s  %s. Sent to %s\n", dbaTime2str(),
```

```
                $alertStatusRef->{$alertType}->{Msg},
                $ref->{Config}->{DBAPager};
      }
      else {
         $ref->{CurrentStatus}->{AlertSent}->{$alertType}->{OK} = 0;
         $ref->{CurrentStatus}->{AlertSent}->{$alertType}->{AlertSentTimeStr}
                        = undef;
      }
    }
    return $ref;
} # alertStatus
```

Table 11-7 describes the command-line arguments accepted by the script `alertScript.pl` in Listing 11-17.

Table 11-7. Command-Line Parameters for the Script in Listing 11-17

PARAMETER	DESCRIPTION
`-C <Cluster name>`	Specifies the name of the cluster to be monitored
`-s <Status file>`	Specifies the file to which the cluster status data structure will be saved and from which the saved status information will be read
`-r <Recipient>`	Specifies the email address of the alert recipient
`-a <SMTP account>`	Specifies the account of the SMTP email sender
`-m <SMTP server>`	Specifies the SMTP mail server

Studying the Cluster Alert Script

The first thing to notice about the script `alertCluster.pl` in Listing 11-17 is that it really has nothing particular to do with SQL Server. In fact, it's devoid of any specific SQL Server knowledge. The script can monitor any Microsoft cluster server irrespective of whether it includes a SQL Server virtual server.

Furthermore, when a cluster group includes a SQL Server virtual server and the SQL Server is offline, the group will be marked `partially offline` or `offline`, which will cause the script in Listing 11-17 to send a notification. If the SQL Server availability is also being monitored—for instance, with the script in Listing 11-13—the DBA will then receive two alerts, one from the cluster monitoring script and one from the availability monitoring script.

These redundant alerts may or may not be palatable to you. There are two schools of thought when it comes to deciding what to monitor and how to notify: the minimalist school and the paranoid school. The minimalist wants to keep the notification to the bare minimum without compromising the effectiveness, and the paranoid welcomes redundancy, treating redundant alerts as corroborating each other and fearing that skimping on alerts may result in an important alert being missed.

If you belong to the minimalist school, you can modify the following code segment in the function `checkStatus()` in Listing 11-17 to skip any cluster group containing a SQL Server virtual server:

```
# check if a group is not online
foreach my $group (@currentGroups) {
   my $status = $currentRef->{Groups}->{$group}->{Status};
   if ($status !~ /^Online$/i) {
      $alertStatusRef->{GroupNotOnline}->{Msg} .= "$group,";
      $ref->{CurrentStatus}->{OK} = 0;
   }
}
```

To find whether a cluster group contains a SQL Server virtual server, you need to enumerate the resources in the group and check each resource to determine whether its resource type is SQL Server service.

Another question about the script `alertCluster.pl` in Listing 11-17 is why it doesn't check the status of any cluster resources. From experiments with monitoring the status changes at the resource level in addition to the node level and the group level, I found that monitoring the resources of a cluster all too often results in overly redundant alerts and significantly complicates the coding of the script.

In addition, note the following:

- In a Microsoft failover cluster, any status change of an important resource almost always bubbles up to the group level.

- Critical resources such as SQL Server and SQL Server Agent are already being monitored, independent of the cluster in which they're running.

All these factors considered, there's little added value to monitor the resources in a cluster when the nodes and the groups are already being monitored.

Finally, notice how the error conditions are collected and alerts are decided. Listing 11-18 shows an example of the current status data structure. The function `checkStatus()` is responsible for populating this data structure.

Listing 11-18. A Sample Cluster Status Data Structure—Single Cluster

```
$ref->{CurrentStatus} = {
   OK => '0',
   ClusterStatus => {
      Nodes => {
             NYCLSQLNODE1 => { 'Status' => 'Up' },
             NYCLSQLNODE2 => { 'Status' => 'Paused' }
      }
      Groups => {
           'ClusterGroup' => {    # group named ClusterGroup
                 Status => 'Online',
                 Node   => 'NYCLSQLNODE2'
           },
            'MSSQL01' => {        # group named MSSQL01
                 Status => 'Online',
                 Node'=> 'NYCLSQLNODE1'
            },
            'MSSQL02' => {        # group named 'MSSQL02'
                 Status => 'Partially Online',
                 Node => 'NYCLSQLNODE1'
            }
      }
   },
   AlertStatus => {
      GroupNotOnline => {
            Msg => 'Cluster NYCLUSTER GroupNotOnline: MSSQL02'
      },
      NodeNotUp => {
            Msg => 'Cluster NYCLUSTER NodeNotUp: NYCLSQLNODE2'
      }
   }
};
```

In the hash record of `$ref->{CurrentStatus}`, there are three keys: `OK`, `ClusterStatus`, and `AlertStatus`. The hash key `OK` is an overall indicator of the cluster status. If it's set to 1, the cluster is in a good condition; there's no need to look at any more detail, and no alert is necessary. If it's set to 0, the function `alertStatus()` will go through the hash record

$ref->{CurrentStatus}->{AlertStatus} to send out an alert for each of the following keys, if its value is defined as follows:

- FailedQuery
- NodeNotUp
- NodeEvicted
- GroupNotOnline
- GroupMoved

In Listing 11-18, by inspecting the information in the hash record $ref->{CurrentStatus}->{ClusterStatus}, the script alertCluster.pl found two events: GroupNotOnline and NodeNotUp. Thus, there are two keys under $ref->{CurrentStatus}->{AlertStatus}, GroupNotOnline and NodeNotUp, to record these two events. The messages associated with these two keys will be sent.

Notice that the keys correspond to the five cluster events this section sets out to monitor. Given the nature of these events, it's possible that more than one of them are detected and recorded in the data structure. For instance, in a cluster, a group can be taken offline while another group is moved to a different node and one node is offline. As a result, the DBA may receive multiple alerts in a row, each on a different cluster event.

To determine whether a node is evicted or a group is moved, the script must compare the current cluster status with the previous cluster status. The previous cluster status is read from the status file and kept in the hash record $ref->{Saved-Status}. Its structure is the same as that of the current cluster status. The code fragment to find whether a node is evicted from the cluster is as follows:

```
# check if a node is evicted
if (@diff = dbaSetDiff(\@savedNodes, \@currentNodes)) {
   my $nodes = join(',', map {"'" . $_ ."'"} @diff);
   $alertStatusRef->{NodeEvicted}->{Msg} = $nodes;
   $ref->{CurrentStatus}->{OK} = 0;
}
```

If you're familiar with the Microsoft failover clustering, the script in Listing 11-17 is almost trivial. However, parsing the output of the utility cluster.exe deserves some explanation. The output of the command cluster node is easy to parse because it neatly conforms to a space-separated, three-column format after skipping the initial header and heading. None of the three columns allow space in its values.

The output of the command `cluster group`, however, can be tricky to parse. Even though its output still consists of three columns, a group name in the first column may contain any number of spaces. Fortunately, because the other two columns can't contain any space, you can get the last two columns first and leave the remainder—whatever it may be—to be the first column.

Monitoring SQL Server Clusters: The Robust Version

> **SCENARIO**
> *Instead of monitoring a single cluster, you need to monitor multiple SQL Server clusters.*

Monitoring multiple SQL Server clusters introduces a host of issues that you must consider. This topic reviews these issues and provides a Perl scripting solution to monitor the important state changes of multiple SQL Server clusters. The solution builds on the script `alertCluster.pl` in Listing 11-17.

Improving SQL Server Cluster Monitoring

The cluster monitoring script `alertCluster.pl` is more a proof of concept than a polished and full-featured implementation for production use. The script `alertCluster.pl` is lacking in the following areas: monitoring multiple clusters, managing configurations, and customizing cluster monitoring. Let's examine what can be improved in these areas in turn.

Monitoring Multiple Clusters

The script `alertCluster.pl` is implemented specifically for monitoring a single cluster. To monitor multiple clusters, you have to put this script in a batch file, calling it once for each cluster. This works with a small number of clusters and becomes unmanageable as the number of clusters increases. Instead of trying to make an inherently single-cluster monitoring tool cope with multiple clusters, you'd be better off with a tool that accommodates multiple clusters by design.

Managing Configurations

Specifying all the configuration options on the command line severely limits the number of options you can practically have. Centrally specifying the options in a configuration file is a better way to go.

Customizing Cluster Monitoring

It's cute to parse the output of the command-line utility `cluster.exe` for the cluster status information. However, when you want to customize the script to accommodate additional monitoring requirements, you often need to retrieve cluster information beyond the simple node and group status, and this approach becomes cumbersome. In most cases, you have to wade through a lot of irrelevant data in the text output to get to that specific piece of information. In addition, because the names for many common cluster constructs such as groups, resources, and resource types can have arbitrary characters including spaces, parsing the output can become rather unsightly. The script will be much easier to customize if you use Cluster Automation Server, which allows you to directly go to an object or a property with the information you want.

NOTE *If you're running Windows 2000, the Cluster Automation Server (*`MSCLUS.DLL`*) is part of Windows 2000, and you don't need to install anything. The best place to find information on programming with the Cluster Automation Server is the Windows Platform Software Development Kit (SDK). The information is also available at the MSDN site (*`http://msdn.microsoft.com`*).*

Creating a Script to Monitor Multiple SQL Server Clusters

With improvements in these areas, the script `monitorClusters.pl` in Listing 11-19 monitors multiple SQL Server clusters for the following seven cluster-related events:

- The cluster isn't accessible.
- A node is evicted from the cluster.
- The status of a node isn't up.
- The status of a group isn't online.
- A group moves from one node to another.
- A group isn't running on its preferred owner node.
- A group has been removed from the cluster.

The first five events are the same as the ones monitored by the script `alertCluster.pl` in Listing 11-17. The requirement for monitoring whether a group is running on its preferred owner node is a practical one. When you have multiple SQL Server instances running in a cluster, a given instance is often configured to run on a specific node—its preferred owner node—under normal circumstances for performance and administrative purposes. Even though the instance works on any node, you want to make sure it's on the preferred owner node, if at all possible.

The script `monitorClusters.pl` in Listing 11-19 monitors multiple SQL Server clusters for these seven events.

Listing 11-19. Monitoring SQL Server Clusters: The Robust Version

```
use strict;
use SQLDBA::Utility qw( dbaSMTPSend dbaTime2str dbaSaveRef dbaSetDiff
                        dbaReadSavedRef dbaSetCommon dbaInSet
                        dbaIsBetweenTime dbaReadINI );
use Win32::OLE qw(in);

Main: {
   my $configFile = shift;
   $configFile or printUsage();
   (-T $configFile) or
         die "***Err: specified config file $configFile does not exist.";

   # Read config file into $configRef
   my $configRef = dbaReadINI($configFile);

   # validate config options and set defaults
   $configRef = validateConfig($configRef);

   # read saved status information from the status file
   my $savedStatusRef =
         (-T $configRef->{CONTROL}->{STATUSFILE}) ?
          dbaReadSavedRef($configRef->{CONTROL}->{STATUSFILE}) : {};

   # merge the config data structure and the status data structure
   my $ref = { Config => $configRef,
               SavedStatus => $savedStatusRef };

   # get current status
   $ref = getStatus($ref);
```

```
   # check the current status and compare it with the saved one
   $ref = checkStatus($ref);

   #  Decide whether to send an alert, and send the alert, if any
   $ref = alertStatus($ref);

   # Save status to the status file
   dbaSaveRef($configRef->{CONTROL}->{STATUSFILE},
              $ref->{CurrentStatus}, 'ref');
} # Main

######################
sub validateConfig {
   my $configRef = shift or die "***Err: validateConfig() expects a reference.";

   foreach my $cluster (sort keys %{$configRef}) {
      next if $cluster =~ /^control$/i;
      next if $configRef->{$cluster}->{DISABLED} =~ /^y/i;

      if (!defined $configRef->{$cluster}->{ALERTINTERVAL} or
          $configRef->{$cluster}->{ALERTINTERVAL} !~ /\d+/) {
          $configRef->{$cluster}->{ALERTINTERVAL} = 15;  # set the default
      }
   }
   return $configRef;
} # validateConfig

#################
sub getStatus {
   my $ref = shift or die "***Err: getStatus() expects a reference.";

   my %nodeState  = ( -1 => 'Unkown',
                       0 => 'Up',
                       1 => 'Down',
                       2 => 'Paused',
                       3 => 'Joining' );
   my %groupState = ( -1 => 'Unkown',
                       0 => 'Online',
                       1 => 'Offline',
                       2 => 'Failed',
                       3 => 'PartialOnline',
                       4 => 'Pending' );

   foreach my $cluster (sort keys %{$ref->{Config}}) {
```

```
next if $cluster =~ /^CONTROL/i;
next if $ref->{Config}->{$cluster}->{DISABLED} =~ /^y/i;

my $statusRef;
my $savedStatusRef = $ref->{SavedStatus}->{$cluster};
my ($clusterName) = $cluster =~ /^CLUSTER\s*:\s*(.+)$/i;

my $clusRef = Win32::OLE->new('MSCluster.Cluster');
unless ($clusRef) {
   my $err = Win32::OLE->LastError();
   $statusRef->{OK} = 0;
   $statusRef->{AlertStatus}->{FailedQuery}->{Times} =
            $savedStatusRef->{AlertStatus}->{FailedQuery}->{Times} + 1;
   $statusRef->{AlertStatus}->{FailedQuery}->{Msg} =
            "Failed new('MSCluster.Cluster'). $err.";
   $ref->{CurrentStatus}->{$cluster} = $statusRef;
   return $ref;
};

$clusRef->Open($clusterName);
if (my $err = Win32::OLE->LastError()) {
   print "***unable to open $cluster. $err\n";
   $statusRef->{OK} = 0;
   $statusRef->{AlertStatus}->{FailedQuery}->{Times} =
            $savedStatusRef->{AlertStatus}->{FailedQuery}->{Times} + 1;
   $statusRef->{AlertStatus}->{FailedQuery}->{Msg} =
            "Failed open($clusterName). $err.";
   $ref->{CurrentStatus}->{$cluster} = $statusRef;
   next;
}

# loop throught the Nodes collection
foreach my $node (in ($clusRef->Nodes)) {
   next unless $node;
   $statusRef->{ClusterStatus}->{Nodes}->{$node->{Name}}->{Status}
                     = $nodeState{$node->{State}};
}

# loop through the ResourceGroups collection
foreach my $group (in ($clusRef->ResourceGroups)) {
   next unless $group;
   my $grpRef = $statusRef->{ClusterStatus}->{Groups}->{$group->{Name}};
   $grpRef->{Status} = $groupState{ $group->{State} };
   $grpRef->{Node} = $group->{OwnerNode}->{Name};
```

```
            # loop through the PreferredOwnerNodes collection
            foreach my $preferredNode (in ($group->{PreferredOwnerNodes})) {
               next unless $preferredNode;
               push @{$grpRef->{PreferredOwnerNodes}},
                  $preferredNode->{Name};
            }
            $statusRef->{ClusterStatus}->{Groups}->{$group->{Name}} = $grpRef;
         }

         $statusRef->{OK} = 1;
         $ref->{CurrentStatus}->{$cluster} = $statusRef;
      }
      return $ref;
   }  # getStatus

   ###################
   sub checkStatus {
      my ($ref) = shift or die "***Err: checkStatus() expects a reference.";

      foreach my $cluster (sort keys %{$ref->{Config}}) {
         next if $cluster =~ /^CONTROL/i;
         next if $ref->{Config}->{$cluster}->{DISABLED} =~ /^y/i;

         my ($clusterName) = $cluster =~ /^CLUSTER\s*:\s*(.+)$/i;
         my $savedRef = $ref->{SavedStatus}->{$cluster}->{ClusterStatus};
         my $curRef = $ref->{CurrentStatus}->{$cluster}->{ClusterStatus};
         my $alertRef;

         my @diff = ();
         my @savedNodes = keys %{$savedRef->{Nodes}};
         my @currentNodes = keys %{$curRef->{Nodes}};
         my @savedGroups = keys %{$savedRef->{Groups}};
         my @currentGroups = keys %{$curRef->{Groups}};

         # check if a node is not Up
         foreach my $node (@currentNodes) {
            if ($curRef->{Nodes}->{$node}->{Status} !~ /^Up$/i) {
               $alertRef->{NodeNotUp}->{Msg} .= "$node,";
               $ref->{CurrentStatus}->{$cluster}->{OK} = 0;
            }
         }

         # check if a group is not online
         foreach my $group (@currentGroups) {
```

```
            my $status = $curRef->{Groups}->{$group}->{Status};
            if ($status !~ /^Online$/i) {
               $alertRef->{GroupNotOnline}->{Msg} .= "$group,";
               $ref->{CurrentStatus}->{$cluster}->{OK} = 0;
            }
         }

         # check preferred owner nodes for each group
         if ($ref->{Config}->{$cluster}->{CHECKPREFERREDOWNERNODES} =~ /^y/i) {
            foreach my $group (@currentGroups) {
               if (defined $curRef->{Groups}->{$group}->{PreferredOwnerNodes}) {
                  unless (dbaInSet($curRef->{Groups}->{$group}->{Node},
                           $curRef->{Groups}->{$group}->{PreferredOwnerNodes})) {
                     $alertRef->{GroupNotOnOwnerNode}->{Msg} .= "$group,";
                     $ref->{CurrentStatus}->{$cluster}->{OK} = 0;
                  }
               }
            }
         }

         # check if a node is evicted
         if (@diff = dbaSetDiff(\@savedNodes, \@currentNodes)) {
            my $nodes = join(',', map {"'" . $_ ."'"} @diff);
            $alertRef->{NodeEvicted}->{Msg} = $nodes;
            $ref->{CurrentStatus}->{$cluster}->{OK} = 0;
         }

         # check if a group is moved to another node
         foreach my $group (dbaSetCommon(\@currentGroups, \@savedGroups)) {
            my $oldNode = $savedRef->{Groups}->{$group}->{Node};
            my $newNode = $curRef->{Groups}->{$group}->{Node};
            if ($oldNode ne $newNode ) {
               $alertRef->{GroupMoved}->{Msg} .= "$group: $oldNode->$newNode,";
               $ref->{CurrentStatus}->{$cluster}->{OK} = 0;
            }
         }
         $ref->{CurrentStatus}->{$cluster}->{AlertStatus} = $alertRef;
      }
      return $ref;
   } # checkStatus

   ##################
   sub alertStatus {
       my $ref = shift or die "***Err: alertStatus() expects a reference.";
```

```
foreach my $cluster (sort keys %{$ref->{Config}}) {
   next if $ref->{CurrentStatus}->{$cluster}->{OK};

   next if $cluster =~ /^CONTROL/i;
   next if $ref->{Config}->{$cluster}->{DISABLED} =~ /^y/i;
   my ($clusterName) = $cluster =~ /^CLUSTER\s*:\s*(.+)$/i;

   my $alertRef = $ref->{CurrentStatus}->{$cluster}->{AlertStatus};
   my $currentRef = $ref->{CurrentStatus}->{$cluster};
   my $savedRef = $ref->{SavedStatus}->{$cluster};
   my @receivers = ( $ref->{Config}->{$cluster}->{DBAPAGER} );

   foreach my $alertType (sort keys %{$alertRef}) {
      # try twice before quitting
      next if (     $alertType eq 'AccessFailed'
                and $alertRef->{AccessFailed}->{Times} <= 2);
      # was an alert sent recently
      next if (time() -
              $savedRef->{AlertSent}->{$alertType}->{AlertSentTime})
                 < ($ref->{Config}->{$cluster}->{ALERTINTERVAL})*60;
      # don't alert if it's in the quiet time period
      next if dbaIsBetweenTime($ref->{Config}->{$cluster}->{QUIETTIME});

      # tidy up the message
      $alertRef->{$alertType}->{Msg} =~ s/,$//;
      $alertRef->{$alertType}->{Msg}
                  = 'Cluster ' . $clusterName .
                    " $alertType: " . $alertRef->{$alertType}->{Msg};
      # send via SMTP
      if (dbaSMTPSend($ref->{Config}->{CONTROL}->{SMTPSERVER},
                      \@receivers,
                      $ref->{Config}->{CONTROL}->{SMTPSENDER},
                      $alertRef->{$alertType}->{Msg})) {

         $currentRef->{AlertSent}->{$alertType}->{OK} = 1;
         $currentRef->{AlertSent}->{$alertType}->{AlertSentTimeStr}
                           = dbaTime2str();
         $currentRef->{AlertSent}->{$alertType}->{AlertSentTime}
                           = time();
         $currentRef->{AlertSent}->{$alertType}->{Msg}
                     = $alertRef->{$alertType}->{Msg};
         # log it to a file
         if (open(LOG, ">>$ref->{Config}->{CONTROL}->{ALERTLOGFILE}")) {
```

```
                printf LOG "%s  %s. Sent to %s\n", dbaTime2str(),
                          $alertRef->{$alertType}->{Msg},
                          $ref->{Config}->{$cluster}->{DBAPAGER};
                close(LOG);
             }
          }
          else {
             $currentRef->{AlertSent}->{$alertType}->{OK} = 0;
             $currentRef->{AlertSent}->{$alertType}->{AlertSentTimeStr}
                               = undef;
             $currentRef->{AlertSent}->{$alertType}->{AlertSentTime}
                               = undef;
          }
       }
       $ref->{CurrentStatus}->{$cluster} = $currentRef;
    }
    return $ref;
} # alertStatus

#################
sub printUsage {
    print << '--Usage--';
Usage:
  cmd>perl monitorClusters.pl <Config File>
       <Config File>   file to specify config options for monitoring clusters
--Usage--
   exit;
}
```

All the configuration options that the script `monitorClusters.pl` accepts are specified in a configuration file. Assuming the configuration file is `config.txt` in the current directory, you can run the script from the command line as follows:

```
cmd>perl monitorCluster.pl config.txt
```

You should schedule to run the script once every few minutes.

Listing 11-20 is a sample configuration file for monitoring two clusters, NYCLSQL01 and NYCLSQL02.

Listing 11-20. A Sample Configuration File for `monitorClusters.pl`

```
[Control]
AlertLogFile=d:\dba\clusters\Alert.log
StatusFile=d:\dba\clusters\status.log
SMTPServer=mail.linchi.com
SMTPSender=sql@linchi.com

[Cluster:NYCLSQL01]
Disabled=no
DBAPager=72001@myTel.com
QuietTime=20-8
AlertInterval=15
CheckPreferredOwnerNodes=yes

[Cluster:NYCLSQL02]
Disabled=no
DBAPager=74321@myTel.com
QuietTime=24-6
AlertInterval=30
CheckPreferredOwnerNodes=no
```

Table 11-8 describes each of these configuration options. For brevity, this table contains both the options in the `Control` section and the options for each cluster.

Table 11-8. Options for `monitorClusters.pl`

OPTION	DESCRIPTION
`AlertLogFile`	A log entry is made to this text file every time an alert is sent.
`StatusFile`	This is the file where the cluster status data structure will be saved. The `monitorClusters.pl` script reads from this file the next time it runs.
`SMTPServer`	The SMTP mail server.
`SMTPSender`	This is the SMTP account to identify the sender of the alert.
`Disabled`	When this is set to `yes`, the script doesn't monitor the cluster.
`DBAPager`	All alerts for this cluster will be sent to this address.

Table 11-8. Options for `monitorClusters.pl` *(Continued)*

OPTION	DESCRIPTION
`QuietTime`	This option accepts two values in the format of hh-hh, where hh is between 1 and 24, inclusive. When this is specified, no alerts for this cluster will be sent between these two hours.
`AlertInterval`	This option specifies the minimum amount of time in minutes that must elapse before another alert for this cluster can be sent again.
`CheckPreferredOwnerNodes`	This is a toggle with values `yes` or `no`. When it's set to `no`, the script won't check whether a group is running on its preferred owner node.

Studying the Script monitorClusters.pl

The overall flow of this script is similar to that of the basic version of the cluster monitoring script `alertCluster.pl` in Listing 11-18, but several salient differences are worth noting.

One such notable difference is between the data structures used by these two scripts. Although the data structure for a given cluster in Listing 11-19 is nearly identical to the data structure used in Listing 11-17, the former has one more level in the overall data structure to explicitly accommodate multiple clusters. Listing 11-21 shows the data structure for the cluster NYCLSQL01 and illustrates the addition of the level immediately below `$ref->{CurrentStatus}` in the data structure hierarchy.

Listing 11-21. A Sample Cluster Status Data Structure—Multiple Clusters

```
$ref->{CurrentStatus}->{'CLUSTER:NYCLSQL01'} = {
      OK => '0',
      ClusterStatus => {
            Nodes => {
                    NYCLSQL01NODE1 => { Status => 'Up' },
                    NYCLSQL01NODE2 => { Status => 'Up' }
             }
            Groups => {
                   MSSQL01 => {
                        PreferredOwnerNodes => [ 'NYCLSQL01NODE1' ],
                        Status => 'Online',
                        Node => 'NYCLSQLNODE2'
                    },
```

```
                    MSSQL02 => {
                         PreferredOwnerNodes => [ 'NYCLSQL01NODE1' ],
                         Status => 'Partial Online',
                         Node => 'NYCLSQL01NODE1'
                     },
                    ClusterGroup => {
                         Status => 'Online',
                         Node => 'NYCLSQL01NODE2'
                     }
               },
          }
         AlertStatus => {
               GroupNotOnPreferredOwnerNode => {
                    Msg => 'Cluster NYCLSQL01 GroupNotOnPreferredOwnerNode: MSSQL01'
               },
               GroupNotOnline => {
                    Msg => 'Cluster NYCLSQL01 GroupNotOnline: MSSQL02'
               }
         },
};
```

Let's compare this data structure with that in Listing 11-17. They're identical with the exception of the additional keys under the individual group to capture the list of preferred owner nodes.

From a programming standpoint, instead of referencing the group `MSSQL01` on the cluster NYCLSQL01 as follows:

```
$statusRef->{ClusterStatus}->{Groups}->{MSSQL01}
```

where `$statusRef` is set to `$ref->{CurrentStatus}`), you now reference it as follows:

```
$statusRef->{'CLUSTER:NYCLSQL01'}->{ClusterStatus}->{Groups}->{MSSQL01}
```

Literally, all you need to do to accommodate multiple clusters is to add another layer of reference (for example, `{'CLUSTER:NYCLSQL01'}`) in all the references and add a loop to go through all the clusters wherever appropriate. This again demonstrates the power of Perl's reference-based dynamic data structure in creating a hierarchical model and the ease with which you can manipulate the hierarchy.

The second salient difference between the script `monitorClusters.pl` in Listing 11-19 and the script in Listing 11-17 is how they implement the function

getStatus(). This function retrieves the node and group status information to populate the current status data structure ($ref->{CurrentStatus} in Listing 11-17 and $ref->{CurrentStatus}->{'CLUSTER:NYCLSQL01'} for the cluster NYCLSQL01 in Listing 11-20). In the script monitorClusters.pl, the function getStatus() employs the Cluster Automation Server to retrieve the status information of the nodes and the status information of the groups in a cluster. The function getStatus() first invokes the Cluster Automation Server as follows to obtain the Cluster object:

```
my $clusRef = Win32::OLE->new('MSCluster.Cluster');
```

Contrast this with the use of the command-line utility cluster.exe in the script alertCluster.pl in Listing 11-17.

The following code fragment in the function getStatus() in Listing 11-19 gets the preferred owner nodes for each group:

```
# loop through the PreferredOwnerNodes collection
foreach my $preferredNode (in ($group->{PreferredOwnerNodes})) {
   next unless $preferredNode;
   push @{$grpRef->{PreferredOwnerNodes}},
      $preferredNode->{Name};
}
```

PreferredOwnerNodes is a property of the Group object, which is a member of the ResourceGroup collection. You can get ResourceGroup from the Cluster object. The preferred owner information would be rather cumbersome to obtain if you rely on parsing the text output of cluster.exe.

Summary

Setting up and maintaining a SQL Server monitoring infrastructure is a never-ending job because it has to accommodate systems and operational changes, and most significantly it has to adapt to organizational changes. Just when you think you've finally hammered in that last nail, you discover that you overlooked a peculiar but still important scenario, you can't ignore an issue as you previously thought, or the evolved environment has given rise to new monitoring requirements. This is true regardless of whether you're customizing a third-party monitoring package or writing your own home-grown monitoring scripts.

The scripts in this chapter demonstrate that writing your own scripts in Perl to effectively monitor SQL Server instances and SQL Server clusters isn't only feasible but that it can also be easily accomplished. These Perl scripts offer ultimate flexibility in meeting new requirements or addressing unanticipated twists in the existing requirements. Furthermore, they were not created in vacuum but have

grown out of acute real-world needs. Scripts similar to the ones discussed in this chapter have been used to successfully monitor substantial SQL Server installations.

Experience suggests that the script for monitoring SQL Server errorlogs, the script for monitoring SQL Server availability, and the script for monitoring SQL Server clusters together comprise a comprehensive set of tools for monitoring exceptions in SQL Server environments.

The next chapter, which is the final chapter of this book, focuses on using Perl to simplify managing SQL Server instances in the enterprise environment.