Microsoft TechNet

Implementing Application Failover with Database Mirroring

SQL Server Best Practices Article

Published: April 19, 2007

Writers: Michael Thomassy, Sanjay Mishra

Technical Reviewers: Prem Mehra, Mike Ruthruff, Don Vilen, Kevin Cox, Chris Lee, Siva Raghupathy, Jaaved Mohammed, Burzin Patel, Cihan Biyikoglu, Madhu Desarapu

Applies To: SQL Server 2005 SP1

Summary: Database mirroring allows a database to fail over from one server to another. For a seamless user experience, it is important that the application automatically reconnect to the current principal server. This paper describes how to enable your application to automatically fail over when database mirroring failover occurs. The providers for client redirect covered in this paper are Microsoft SQL Native Client, ADO.NET, and Microsoft JDBC Driver.

On This Page

Introduction

Database mirroring allows a database to fail over from one server to another. For a seamless user experience, it is important that the application automatically reconnect to the new database server. This paper describes how to enable your application to automatically fail over when database mirroring failover takes place.

When database mirroring initiates a failover from one partner to the other, it is necessary that the client application reconnect to the new principal server. Once the application successfully reconnects to the new principal server, it can continue its work. Database mirroring failover handles failing over the database to the new principal server instance. To achieve application failover, the client application must redirect a new connection to the current principal, which is a different server instance. The client redirect logic enables an application to be redirected to the current principal server when establishing a new connection. The client redirect logic is implemented in three Microsoft® providers. The three providers discussed in this paper are Microsoft SQL Native Client (SQL Native Client), ADO.NET 2.0, and Microsoft SQL Server 2005 JDBC Driver.

An application programming best practice is to catch and handle errors when an application is connecting to a database and executing statements against a database. By leveraging the client redirect functionality, an application can achieve automatic application failover by implementing retry logic that catches the errors during fail over, then opens a new database connection and retries its queries or transactions.

Top of page

Database Mirroring Failover Operating Modes

The database mirroring feature in Microsoft SQL Server™ 2005 SP1 provides three modes of operation:

- High-safety mode with a witness (synchronous)
- High-safety mode without a witness (synchronous)
- High-performance mode (asynchronous)

High-safety mode with a witness allows a database to fail over from one partner to another either automatically or manually. High-safety mode without a witness allows only manual failover of the database. In high-performance mode, failover can be achieved by using FORCE_SERVICE_ALLOW_DATA_LOSS. (The best practice recommendation is to change the safety mode to FULL (synchronous), do a manual failover, and then put the safety mode back to OFF (asynchronous), wherever applicable, to avoid data loss.) Using the supported client providers (discussed in the next section), an application can take advantage of database mirroring failover and redirect new client connections to the new principal upon database failover. Client redirection works with any of the three operating modes of database mirroring.

For more information

For more details on database mirroring, see:

- Database Mirroring in SQL Server 2005 Books Online
- Database Mirroring in SQL Server 2005 white paper on Microsoft TechNet
- Database Mirroring Best Practices and Performance Considerations white paper on TechNet

Top of page

Providers Supporting Database Mirroring Client Redirect

Three providers have client redirect functionality:

- SQL Native Client
- ADO.NET 2.0 provider for SQL Server 2005 (SqlClient)
- SQL Server 2005 JDBC (Java Database Connectivity) 1.1 Driver

SQL Server 2005 provides new data access technology in SQL Native Client, that includes database mirroring client redirect functionality. SQL Native Client functionality is included in a DLL (sqlncli.dll) with the SQL Native Client ODBC driver and the SQL Native Client OLE DB provider. To use SQL Native Client as a data provider, applications can be written in the .NET Framework (managed code) using the ODBC or OLE DB managed providers where they specify the SQL Native Client provider. Native applications can be written by using ADO or the native ODBC or OLE DB APIs where the appropriate SQL Native Client provider is specified. ADO.NET 2.0 is part of the .NET Framework 2.0 and managed applications program against the **System.Data.SqlClient** namespace. Java applications built with the JDK 1.4.2.12 may use the SQL Server 2005 JDBC 1.1 driver.

For the remainder of this document, whenever SQL Native Client is the only provider mentioned, the information also applies to all three providers discussed in this paper.

Top of page

Automatic Failover Client

SQL Native Client implements the client redirect logic when connecting to a mirrored database. The client redirect logic transparently redirects connection attempts to the current principal server instance, even after a database failover.

After the application successfully reconnects to the new principal server, it can restart any transactions that may have failed because of the broken connection. For a client application to achieve automatic failover with database mirroring, it requires:

- Connectivity with one of the three providers that support database mirroring client redirect.
- A connection string specifying a database.
- A connection string specifying the second partner server instance.
- Application retry logic to handle errors during automatic failover.

Failover Partner

The failover partner is the partner server to use if the connection to the primary server fails. The failover partner can be obtained by two methods. In the first method, SQL Native Client caches the failover partner server name when connecting to the primary server. In the second method, the failover partner is explicitly specified in the connection string.

Cached Failover Partner

When a client that is using SQL Native Client first successfully connects to the principal database in a database mirroring session, the mirror server name and instance name are cached in SQL Native Client. The mirror name is cached even if the failover partner name is not specified in the connection string. The application is redirected to the mirror after an automatic failover.

However, an application that tries to make a new connection to the original principal after an automatic failover has taken place, will fail to connect. The reason for this failure to connect is that SQL Native Client was never connected to the original principal and never cached the failover partner name. Therefore, the recommendation is to explicitly specify the failover partner server in the connection string. The SQL Native Client redirect logic will then connect to the current principal server.

Connection String

The SQL Native Client data providers introduced a new connection string keyword to support some of the new features in SQL Server 2005. The failover partner keyword is used to specify the second partner of the database mirroring session in the connection string. SQL Native Client connects to whichever server is the principal at the time the connection is made.

Following is an example connection string:

```
Data Source=SQLA\INST1;Failover Partner=SQLB\INST1;Initial Catalog=DBMTest;Integrated Security=True
```

For more information on connection attributes when using SQL Native Client with ADO.NET, OLE DB, and ODBC, see Using Connection String

Keywords with SQL Native Client in SQL Server 2005 Books Online.

The syntax of the failover partner keyword is slightly different from one connection method to another:

- OLE DB: `FailoverPartner` (no space between "Failover" and "Partner")
- ODBC: `Failover_Partner` (underscore between "Failover" and "Partner")
- ADO.NET: `Failover Partner` (one space between "Failover" and "Partner")
- JDBC: `failoverPartner` (no space between "failover" and "Partner")

Note: Keywords are not case sensitive. Make sure there is no white space before or after the keyword, before or after the semi-colons, and before or after the equal to operator in the connection string.

Application Retry Logic

The application retry logic is designed to catch any errors that occur while connecting to the database or executing commands (queries or transactions) on the database. When an error occurs, connectivity is re-established, and then failed commands are re-executed if necessary (not all failures mean the transaction failed). As discussed previously, the failover partner should be specified in the connection string so that SQL Native Client can redirect the client connection to the new principal server.

Open New Connections

A database mirroring failover may be initiated by a number of means. For example, a database administrator (DBA) initiates a failover, a network error occurs, a SQL Server instance is taken offline, the partner server goes offline, and many other reasons.

Any of these cases and others will break all existing client connections to the principal database. All applications must close their connections because they are broken, and then open new connections. The time required to open a new connection depends on the time for the database mirroring failover process to complete and for the database to be available.

After an automatic failover from one database partner to another, SQL Native Client redirects all new connections to the current principal. Existing connections are broken and must be closed, cleaned up, and reopened.

During a database failover, transactions that did not complete successfully before failover are automatically rolled back to maintain data integrity (ACID-atomicity, consistency, isolation, durability). The application must retry any failed transactions after a new connection is successfully established.

Retry Logic Flow

The term *retry* implies that the application can re-execute a command against the database upon a failure. A failure could occur for many reasons, but the assumption here is that the failure is due to a failover in the database mirroring session where the application was executing a command. Retry also implies that the application attempts to re-execute the command a certain number of times before giving up or *failing* completely. *Retrying a command* refers to re-executing the complete set of work that was attempted but not completed. The following pseudo-code outlines one approach to automatic failover logic:

1. Enter the retry loop in case an error occurs.
2. Open the database connection.
3. Execute the command.
4. If there were no errors, clean up, and then **return success.**
5. Catch Error on Open Connection or Execute Command.
6. Clean up the **Command** and database **Connection** objects.
7. If the maximum number of retries have not yet been exceeded:
    a Sleep and continue at step 1.
    b Else **return failure.**

Being pseudo-code, the logic is simplified for the purposes of discussion and does not include error handling. Since a database failover causes the connection to break, errors resulting in broken connectivity are not just database errors, but may also be due to network errors, which will result in a connection retry. All retries start with closing the existing connection and attempting to successfully open a new database connection.

The number of retries and how often they occur is application dependent. The number of retries are counted as the number of open database connection attempts. If the maximum number of retries are exceeded, the connection attempt is deemed to have failed. Applications have different ways of handling failed commands, such as logging the error. Since all client failover errors result in a new connection attempt, there should be a specified time to wait (sleep) before attempting another connection. This wait time is called the *retry interval*.

Validate Transaction on Failure

There are cases, independent of database mirroring, where an error is received by the application indicating some sort of failure; however, the database successfully commits the transaction. This can happen if the database successfully committed the transaction, but the connection breaks before the database returns a "transaction successful" message to the client. So, the client reacts as if the transaction has failed (because it hasn't received a confirmation from the database), whereas the database has already successfully committed the transaction.

In this example, if the database is set up with database mirroring in high-safety mode, the transaction is committed to both partners before returning to the client; however, if database failover is initiated, the client may not receive a "transaction successful" message and may instead receive an error.

The application retry logic for transactions needs to account for these rare cases. Otherwise, the transaction might be committed to the database twice—once before the fail over and once after the fail over. Upon a successful reconnect, the retry logic should do an additional check for the existence of the transaction or transactions before executing them again. This may not always be possible since not all transactions are unique, so there is the potential for duplicates. Again, this is possible in any application retry logic independent of database mirroring. Database table-level constraints, such as primary key and unique key, can come in handy to prevent duplicates in the database.

Top of page

Implementing Automatic Application Failover

Automatic application failover can be implemented with managed or native code by using Microsoft C#® .NET, Visual Basic® .NET, Visual Basic, Visual C++® .NET or any number of languages with the SQL Native Client data providers. The sample code provided here is written in the .NET Framework 2.0 and Visual C#. The connection strings when using SQL Native Client with ADO.NET, OLE DB, ODBC, and JDBC use slightly different keywords as documented in SQL Server Books Online.

Database Mirroring Setup

The environment used to test the following example code consisted of three systems installed with Microsoft Windows Server™ 2003 Enterprise Edition with SP1, and SQL Server 2005 Enterprise Edition with SP1.

Role	Name	Instance	DB Name	Port
Principal	SQLA	INST1	DBMTest	5022
Mirror	SQLB	INST1	DBMTest	5022
Witness	SQLW	Default	-	5022

ADO.NET Code Example

This code example uses ADO.NET 2.0 (SqlClient). The following parameters are used in the example.

- `strConn` is the database connection string that includes the failover partner.
- `strCmd` is the command, such as Transact-SQL or a stored procedure, to execute.
- `iRetryInterval` is the number of seconds to wait (sleep) between retries.
- `iMaxRetries` is the number of times to retry the command before failing.

Example connection string (strConn)

```
Data Source=SQLA\INST1;Failover Partner=SQLB\INST1;Initial Catalog=DBMTest;Integrated Security=True
```

```
using System.Data;
using System.Data.SqlClient;
using System.Threading;

bool ExecuteSQLWithRetry_NoResults(string strConn,
                                   string strCmd,
                                   int iRetryInterval,
                                   int iMaxRetries)
{
    // Step 1: Enter the retry loop in case an error occurs
    for (int iRetryCount = 0; iRetryCount < iMaxRetries; iRetryCount++)
    {
        try
        {
            // Step 2: Open the database connection
            conn = new SqlConnection(strConn);
            conn.Open();
            // Step 3: Execute the command
            if (null != conn && ConnectionState.Open == conn.State)
            {
                cmd = new SqlCommand(strCmd, conn);
                cmd.ExecuteNonQuery();
                // Step 4: If there were no errors, clean-up & return success
                return true;
            }
        }
        catch (Exception ex)
        {
            // Step 5: Catch Error on Open Connection or Execute Command
            // Error Handling to be added here: ex
```

```
        }
        finally
        {   // Step 6: Clean up the Command and database Connection objects
            try
            {
                // Clean Command object
                if (null != cmd)
                    cmd.Dispose();
                cmd = null;
                // Clean up Connection object
                if (null != conn && ConnectionState.Closed != conn.State)
                    conn.Close();
                conn = null;
            }
            catch (Exception ex)
            {
                // Error handling to be added here
            }
        }
        // Step 7: If the maximum number of retries not exceeded
        if (iRetryCount < iMaxRetries)
        {
            // Step 7a: Sleep and continue (convert to milliseconds)
            Thread.Sleep(iRetryInterval * 1000);
        }
    } // end for loop: iRetryCount < iMaxRetries
    // Step 7b: If the max number of retries exceeded, return failure
    return false;
}
```

JDBC Code Example

The Java code example using JDBC is almost identical to the C# version with the steps listed in the code. This example was built and executed with the JDK 1.4.2.12 and SQL Server JDBC Driver version 1.1.1501.101. The following parameters are used in the sample code.

- `strConn` is the database connection string that includes the failover partner.
- `strCmd` is the command, such as Transact-SQL or a stored procedure, to execute.
- `iRetryInterval` is the number of seconds to wait (sleep) between retries.
- `iMaxRetries` is the number of times to retry the command before failing.

Example connection string (strConn)

```
jdbc:sqlserver://SQLA;instancename=INST1;failoverPartner=SQLB\INST1;databaseName=DBMTest;integratedsecurity=true
```

```
import java.sql.*;

boolean ExecuteSQLWithRetry_NoResults(String strConn,
                                      String strCmd,
                                      int iRetryInterval,
                                      int iMaxRetries)
{
    // Declare the JDBC objects
    Connection conn = null;
    Statement stmt = null;
    // Step 1: Enter the retry loop in case an error occurs
    for (int iRetryCount = 0; iRetryCount < iMaxRetries; iRetryCount++)
    {
        try
        {
            // Step 2: Open the Database Connection
            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
            conn = java.sql.DriverManager.getConnection(strConn);
            // Step 3: Execute the command
            if (null != conn)
            {
                stmt = conn.createStatement();
                if (null != stmt)
                {
                    stmt.executeUpdate(strCmd);
                    // Step 4: If no errors, clean up & return success
                    return true;
                }
            }
        }
        catch (Exception ex)
        {
            // Step 5: Catch Error on Open Connection or Execute Command
            // Error handling to be added here: ex
        }
        finally
        {  // Step 6: Clean up the command and Database Connection objects
```

```
            try
            {
                // Clean Statement object
                if (null != stmt)
                    stmt.close();
                stmt = null;
                // Clean up Connection object
                if (null != conn)
                    conn.close();
                conn = null;
            }
            catch (Exception ex)
            {
                // Error handling to be added here
                stmt = null;
                conn = null;
            }
        }
        // Step 7: If the maximum number of retries not exceeded
        if (iRetryCount < iMaxRetries)
        {
            // Step 7a: Sleep and continue (convert to milliseconds)
            Thread.sleep(iRetryInterval * 1000);
        }
    } // end for loop: iRetryCount < iMaxRetries
    // Step 7b: If the max number of retries exceeded, return failure
    return false;
}
```

Top of page

Conclusion

Database mirroring failover takes care of failing over the database to the new principal server instance. To achieve application failover, the client application must redirect a new connection to the current principal, which is a different server instance. The client redirect logic is implemented in three Microsoft providers: SQL Native Client, ADO.NET, and JDBC. The client redirect logic enables an application to be redirected to the current principal server when establishing a new connection. By taking advantage of client redirect functionality, an application can achieve automatic application failover by implementing retry logic that catches errors during failover, and then opens a new database connection and retries queries or transactions.

Top of page

Appendix: Hardware Environment

Database Servers

One Unisys ES7000-600 server was partitioned into two servers to act as two partners in the database mirroring setup. Each server consists of:

- Four processors @3.3 MHz, Potamic class, Single core, 1-MB L2, 8-MB L3
- 8-GB RAM
- One Network card, 1-Gbs (gigabits per second)
- One host bus adapter (HBA), 2-Gbs fiber channel

Storage

For storage, a Storage Area Network (SAN) was used with the following configuration on each server:

- NEC S4300
- Cache: 16 gigabyte (GB) mirrored
- Disks: 33 GB @ 15000 RPM, as in the following layout:

Purpose	Drive	RAID	# LUNs	Disks/LUN	Total GB
Log	L:	10	1	8	132
Data	M:	10	1	8	132

Top of page